(4)



An Annual Progress Report
Contract No. N00014-86-K-0245
September 1, 1987 - October 1, 1988

# THE STARLITE PROJECT

Applied Math and Computer Science
Dr. James G. Smith, Program Manager, Code 1211

Computer Science Division
Dr. Andre van Tilborg, Scientific Officer, Code 1133

Submitted to:

Director
Naval Research Laboratory
Washington, D.C. 20375

Attention: Code 2627

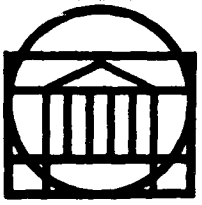
Submitted by:

R. P. Cook
Associate Professor

S. H. Son
Assistant Professor

Report No. UVA/525410/CS89/102
September 1988





SCHOOL OF ENGINEERING AND
APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE



UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

An Annual Progress Report
Contract No. N00014-86-K-0245
September 1, 1987 - October 1, 1988

# THE STARLITE PROJECT

Applied Math and Computer Science
Dr. James G. Smith, Program Manager, Code 1211

Computer Science Division
Dr. Andre van Tilborg, Scientific Officer, Code 1133

Submitted to:

Director
Naval Research Laboratory
Washington, D.C. 20375

Attention: Code 2627

Submitted by:

R. P. Cook
Associate Professor

S. H. Son
Assistant Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA



Report No. UVA/525410/CS89/102
September 1988

Copy No. _____

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>Unclassified | 1b. RESTRICTIVE MARKINGS<br>None |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION / AVAILABILITY OF REPORT<br>Approved for Public Release; Distribution Unlimited |
| 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>UVA/525410/CS89/102 | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION<br>University of Virginia<br>Department of Computer Science<br>6b. OFFICE SYMBOL (If applicable) | 7a. NAME OF MONITORING ORGANIZATION<br>Office of Naval Research Resident Representative |
| 6c. ADDRESS (City, State, and ZIP Code)<br>Thornton Hall<br>Charlottesville, VA 22901 | 7b. ADDRESS (City, State, and ZIP Code)<br>818 Connecticut Ave., N.W., Eighth Floor<br>Washington, DC 20006 |
| 8a. NAME OF FUNDING / SPONSORING ORGANIZATION<br>Office of Naval Research<br>8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER<br>N00014-86-K-0245 |
| 8c. ADDRESS (City, State, and ZIP Code)<br>800 N. Quincy Street<br>Arlington, VA 22217-5000 | 10. SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |
| 11. TITLE (Include Security Classification)<br>The Starlite Project | |
| 12. PERSONAL AUTHOR(S)<br>R. P. Cook, S. H. Son | |
| 13a. TYPE OF REPORT<br>Annual Progress<br>13b. TIME COVERED FROM 09/01/87 TO 10/01/88 | 14. DATE OF REPORT (Year, Month, Day)<br>1983, September<br>15. PAGE COUNT<br>43 |
| 16. SUPPLEMENTARY NOTATION | |

| 17 COSATI CODES | | |
|---|---|---|
| FIELD | GROUP | SUB-GROUP |
| | | |
| | | |

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The StarLite Project has the goal of constructing a program library for real-time applications. The initial focus of the project is on operating system and database support. The project also involves the construction of a prototyping environment that supports experimentation with concurrent and distributed algorithms in a host environment before down-loading to a target system for performance testing.

The components of the project include a Modula-2 compiler, a symbolic Modula-2 debugger, an interpreter/runtime package, the Phoenix operating system, the meta-file system, a visual simulation package, a database system, and documentation.

| | |
|---|---|
| 20. DISTRIBUTION / AVAILABILITY OF ABSTRACT<br>☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION<br>Unclassified |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL<br>Dr. James G. Smith | 22b. TELEPHONE (Include Area Code)<br>(202) 696-4713<br>22c. OFFICE SYMBOL |

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

TABLE OF CONTENTS

## 1. Introduction

It seems improbable that a single operating system will suffice to solve all the application problems that are likely to arise in future real-time, embedded systems. A much more likely scenario is that future engineers, with support from a programming environment, will select and adapt modules from program libraries. The selected modules must have proven operating characteristics and the domain over which they are applicable must be well-defined.

The StarLite Project, which is supported by the Office of Naval Research, has the goal of constructing such a program library for real-time applications. The initial focus of the project is on operating system and database support.

Another goal of the StarLite project is to test the hypothesis that a host prototyping environment can be used to significantly accelerate our ability to perform experiments in the areas of operating systems, databases, and network protocols. The primary project requirement for StarLite is that software developed in the prototyping environment must be capable of being retargeted to different architectures only by recompiling and replacing a few low-level modules. The anticipated benefits are fast prototyping times, greater sharing of software in the research community, and the ability for one research group to validate the claims of another by replicating experimental conditions exactly.

As one measure of the effectiveness of the environment, it is often possible to fix errors in the operating system, compile, and reboot the StarLite virtual machine in less than twenty seconds. The compilation time on a SUN 3/280 for the 66 modules (7500 lines) that comprise the operating system is one minute (clock) or 16 seconds (user time). The StarLite VM, as measured by Wirth's Modula-2 benchmark program[1], executes at a speed of from one to six times that of a PDP 11/40, depending on the mix of instructions.

The StarLite prototyping architecture is designed to support the simultaneous execution of multiple operating systems in a single address space. For example, to prototype a distributed operating system, we might want to initiate a file server and several clients. Each virtual machine would have its own operating system and user processes. All of the code and data for all of the virtual machines would be executed as a single UNIX process.

In order to support this requirement, we assume the existence of high-performance workstations with large local memories. Ideally, we would prefer multi-thread support, but multiprocessor workstations are not yet widely available. We also assume that hardware details can be isolated behind high-level language interfaces to the extent that the majority of a system's software remains invariant when retargeted from the host to a target architecture.

The progress to date and a brief description of future work for each of the StarLite components are listed in Figure 1. Each component of the project is covered in greater detail in later Sections. At the present time, all components execute on SUN workstations using the StarLite Modula-2 system.

## 2. Related Activities

Cook and Son, participants in the IBM Manassas Real-Time Workshop, (April 1988).
Cook and Son, participants in coordination meeting with Professor Tokuda from CMU and Pat Watson from IBM, (July 1988).
Cook and Son, participants in coordination meeting with Pat Watson from IBM, (August 1988).
Cook, reviewed the Draft NATO Requirements and Design Criteria for the NATO Standard Interface Specification of Ada Programming Support Environments at the request of LTC(P) David R. Taylor (AJPO), (Feb. 1988).
Cook, Session chair and panel member, Fifth IEEE Workshop on Real-Time Software and Operating Systems, (May 1988).

Figure 1. PROGRESS TO DATE FOR EACH COMPONENT OF STARLITE AND FUTURE PLANS

| | | | |
|---|---|---|---|
| COMPILER | 1/1/88 - Converted one-pass compiler to C for portability. | 6/1/88 - Ported compiler to the SUN environment. | Extend support for reuseability. Retarget a C compiler to the prototyping architecture. |
| DEBUGGER | 1/1/88 - Started rewrite of debugger to be more generic. | 6/1/88 - Finished version one of the debugger. | |
| INTERPRETER | 1/1/88 - Converted runtime to C for portability. | 6/1/88 - Ported runtime to the SUN; added a multiprocessor virtual machine module; revised architecture to improve machine independence. | Extend options on integrity checking to improve error detection coverage. |
| PHOENIX OS | 1/1/88 - Phoenix rewritten to minimize interrupt response time and to improve extensibility. | 6/1/88 - Added a filesystem and a shell to Phoenix. | Modify Phoenix to explore the trade-offs between functionality and real-time guarantees. Investigate automatic configuration generation. |
| DATABASE SYSTEM | 1/1/88 - Performance testing of multi-version database modules. | 7/1/88 - Ported system to StarLite; revised implementation to improve modularity; experimented with priority ceiling algorithms. | Explore trade-offs between functionality and real-time guarantees. Integrate the OS and DB systems. Investigate failure issues. |
| DOCUMENTATION | 1/1/88 - Completed manual for the compiler. | 8/1/88 - Completed architecture manual. | Document system and organize modules for easy retrieval. |

Son, participant San Jose ONR Real-Time Initiative Workshop, (Dec. 1987).
Son, Guest Editor, ACM SIGMOD Record, Special Issue on Real-Time Database Systems, (March 1988).

### 3. Student Participation

Chun-Hyon Chang (Post Doc.), priority-based contention protocol
Veena Bansal (Ph.D. student), debugger
Jeremiah Ratner (Ph.D. student), database development
Yumi Kim (M.S.), multi-version database evaluation
Chris Koeritz(M.Sc. student), operating system
Richard Testardi(M.Sc.), compiler
Jenona Whitlach(M.S.), file system development
Nancy Yeager(M.S.), meta-file system
Richard McDaniel(B.S. student), prototyping environment

### 4. Publications

#### • Journal Publications

(1) Cook, R. P. and S. H. Son, "StarLite, A Software Prototyping Environment," *IEEE Computer Special Issue on Rapid Prototyping*, (submitted).

(2) Cook, R. P., "An Empirical Analysis of the Lilith Instruction Set," *IEEE Transactions on Computers*, (to appear).

(3) Son, S. H., "Replicated Data Management in Distributed Database Systems," *ACM SIGMOD Record*, (to appear).

(4) Son, S. H., "Semantic Information and Consistency in Distributed Real-Time Systems," *Information and Software Technology 30*, September 1988 (to appear).

(5) Son, S. H., "Real-Time Database Systems: Issues and Approaches," *ACM SIGMOD Record, Special Issue on Real-Time Database Systems 17*, 1, March 1988.

(6) Son, S. H., "Using Replication to Improve Reliability in Distributed Information Systems," *Information and Software Technology 29*, October 1987.

#### • Refereed Conference Publications

(7) Cook, R. P., "The StarLite Prototyping Architecture," *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (submitted).

(8) Shebalin, P., S. H. Son, and C.-H. Chang, "An Approach to Software Safety Analysis in a Distributed Real-Time System," *Third Annual Conference on Computer Assurance (COMPASS '88)*, Washington, DC, July 1988, pp 29-43.

(9) Son, S. H., "A Message-Based Approach to Distributed Database Prototyping," *Fifth IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, May 1988, pp 71-74.

(10) Son, S. H., "An Adaptive Checkpointing Scheme for Distributed Databases with Mixed Types of Transactions," *Fourth International Conference on Data Engineering*, Los Angeles, California, Feb. 1988, pp 528-535.

(11) Son, S. H. and J. L. Pfaltz, "Reliability Mechanisms for ADAMS," *Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, California, January 1988.

(12) Son, S. H., "Efficient Decentralized Checkpointing in Distributed Database Systems," *21st Hawaii International Conference on System Sciences*, Kailua-Kona, Hawaii, Jan. 1988, Vol. 2, pp 554-560.

(13) Son, S. H., "Using Replication for High Performance Database Support in Distributed Real-Time Systems," *8th IEEE Real-Time Systems Symposium*, San Jose, California, Dec. 1987, pp 79-86.

(14) Son, S. H., "A Recovery Scheme for Database Systems with Large Main Memory," *11th International Computer Software and Applications Conference (COMPSAC 87)*, Tokyo, Japan, Oct. 1987, pp 422-427.

- **Technical Reports**

(15) Cook, R. P., "Minimizing Response Time".

(16) Cook, R. P., "An Introduction to Modula-2 for Pascal Programmers".

(17) Cook, R. P., "An Introduction to Modular Programming".

(18) Son, S. H. and Y. Kim, "A Prototyping Environment for Distributed Database Systems," *Technical Report TR-88-20*, Dept. of Computer Science, University of Virginia, August 1988.

(19) Shebalin, P., S. H. Son, and C.-H. Chang, "An Approach to Software Safety Analysis in a Distributed Real-Time System," *Technical Report TR-88-13*, Dept. of Computer Science, University of Virginia, May 1988.

(20) Son, S. H. and S. Tripathi, "Distributed Database Systems: Failure Recovery Procedure," *Technical Report TR-88-6*, Dept. of Computer Science, University of Virginia, March 1988.

## 5. The Prototyping Environment

The components of the environment include a Modula-2 compiler, a symbolic debugger, an interpreter/runtime package, the Phoenix operating system, a visual simulation package, and documentation. With Professor Davidson's help, we will eventually retarget his C compiler to produce code for the interpreter/runtime package. Thus, either C or Modula-2 can be used for development. The environment, which currently runs on PCs and SUNs, will also be portable to other hosts.

The programming environment at present consists of a Modula-2 one-pass compiler, interpreter, and simulation package. The compiler supports the Revised Modula-2 Language Definition, except for the LONGREAL/CARD types; LONGINT is supported. Its compilation

speed is twice as fast as the Logitech 286 compiler and five times as fast as the SUN-3 Modula-2 compiler. It also compiles faster than either the MicroSoft C compiler on a PC286 or the SUN-3 C compiler. Fast compilation has been rated as essential to the success of a programming environment (see, for example, Xerox CSL-80-10).

Both the compiler and runtime were ported to C during the past year. The compiler is also implemented in Modula-2. The generated code is for a 32-bit virtual architecture(S-code) that is designed to be extremely space efficient. For example, the object code sizes for a program consisting of 1,000 assignment statements was SUN-Modula(130K), SUN-C(65K), PC286-C(35K), PC286-Modula(11K). Compact code has a significant effect on the speed with which the environment can load both system components and user-level programs that might run on those components. Code generators for a number of target environments are planned for the future. In fact, Professor Davidson had a MS student retarget the Modula-2 back-end to the VAX over the summer to demonstrate the feasibility of this claim.

The interpreter/runtime system for the environment is unique in a number of respects. First, it supports dynamic linking; that is, modules are loaded at the point that one of their procedures is called. Thus, a large software system begins execution very quickly and then loads only the modules that are actually tested. At the current time, a linker is superfluous; as soon as a module is compiled, it may be executed. The second feature of the interpreter is that it maximizes sharing. There will be only one copy of shared code no matter how many times it is used at either the user or operating system levels. Next, the clocks on the interpreter's virtual machines are driven by the number of S-code instructions executed or the actions of simulated devices. Thus, timings for the StarLite host environment can be used to approximate those in a target environment by varying the ratio of S-code instructions necessary for a clock tick. Finally, the interpreter is designed to support a number of different execution models.

During the past year, we completed the implementation of the distributed processor model together with a window package that provides a virtual terminal for each processor. To date, we have tested it with six nodes, each executing UNIX.

The visual simulation package incorporates many of the features of the GPSS simulation language. The traditional "delay" function is provided, as well as the Store and Table simulation types that are used for statistics gathering. Typically, the presence of simulation code is isolated at the lowest levels of a system. By keeping interfaces compatible, a simulation module can be replaced by a module for the target machine. Thus, the higher levels of the software hierarchy remain unchanged when moving code from the host environment to a target.

The final component of the environment is the symbolic debugger, which was also completed during the past year. It allows the user to name any component of a running program and to retrieve the value, type, or address associated with any name. Also, it has the capability to examine multiple threads of control. Eventually, we will add support for user-defined "views" of data abstractions and the ability to view data other than program images. For example, the debugger could be used to examine, or modify, a file that was described in the Modula-2 Interface Definition Language, which is supported by the compiler.

In summary, the environment is designed to maximize productivity. Therefore, it accelerates a researcher's ability to conduct experiments, which advances the state-of-the-art. While the initial version of the environment executes as a single UNIX process, future versions could take excellent advantage of both load balancing to distribute a running prototype across a number of machines and of multiprocessor support, such as is found in Mach or Taos.

## 6. Operating System

During the past year, additional functionality was added to the Phoenix operating system, including a file system and a shell. The system is unique in that it is object-based and is

implemented as a module hierarchy. Also, Phoenix attempts to minimize the use of shared locks, which is the opposite of most current UNIX implementations. Phoenix also uses an integrated priority mechanism that is applied uniformly across all system queues.

Our goal for the coming year is to add real-time guarantees to the Phoenix system. If we are successful, it will be the first such UNIX system. We will also be working to extend the functionality of the operating system as we do not currently support all of SVID. In situations where guarantees cannot be provided, we will change the UNIX interface.

During the past year, we also tested a distributed six-node Phoenix system. However, since we haven't implemented network device drivers, the nodes could not communicate. We will be adding the drivers during the coming year so that we can experiment with real-time issues in the distributed programming area.

Another goal for the coming year is to integrate the Phoenix OS and the database system. At the present time, the database research uses simulated file systems. By using Phoenix, real file systems can be used at each node. As a result, any real-time guarantees that we can provide will be across a complete, end-to-end, distributed operating environment.

## 7. Database Systems

It has been recognized that database systems are assuming much greater importance in real-time systems, which must maintain high reliability and high performance. State-of-the-art database systems are typically not used in real-time applications due to two inadequacies: poor performance and lack of predictability. Current database systems do not schedule their transactions to meet response requirements and they commonly lock data tables indiscriminately to assure database consistency. Locks and time-driven scheduling are basically incompatible. Low priority transactions can and will block higher priority transactions leading to response requirement failures. New techniques that are compatible with time-driven scheduling and the system response predictability need to be investigated.

Our research effort during the past year was concentrated in two areas: investigating new techniques for real-time database systems and developing a message-based database prototyping environment. In addition, to avoid the useless effort in "re-inventing the wheel", Professor Son has spent a significant amount of effort working as a guest editor for the *ACM SIGMOD Record*, collecting current research work in real-time database systems being investigated by other researchers. Selected papers were published in the *ACM SIGMOD Record* special issue on **Real-Time Database Systems** (Vol. 17, No. 1, March 1988). This was the first time that a whole issue of the *ACM SIGMOD Record*, a publication widely circulated in the database research community, has been devoted to a special topic. This was also the first time that research work in real-time database systems was collected and published in a single issue.

### 7.1. New Approaches

Compared with traditional databases, real-time database systems have a distinct feature: they must satisfy not only the database consistency constraints but also the timing constraints associated with transactions. In other words, "time" is one of the key factors to be considered in real-time database systems. Transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. For example, both the update and query on a tracking data of a missile must be processed within the given deadlines: otherwise, the information provided could be of little value.

We have investigated two approaches in designing real-time database systems. The first approach is to redesign a conventional database system's architecture to replace bottleneck components (e.g., a disk) by a high-speed version. A main-memory database system falls in this category. The second approach is to trade desired features (such as serializability) for higher

performance or to exploit semantic information of transactions and data to use the notion of correctness different from the serializability of transaction execution. This approach, combined with effective use of data replication may improve performance and reliability.

The availability of large, relatively inexpensive main memories coupled with the demand for faster response time for real-time database systems has brought a new perspective to database system designers: main memory databases in which the primary copies of all data reside permanently in main memory. Since database operations are mostly I/O bound, elimination of disk access delays can contribute to substantial improvement in transaction response time. However, the migration of data from secondary storage to main memory requires a careful investigation of the components of traditional database management systems, since they introduce some potential problems of their own. The most critical problem is associated with the recovery mechanism of the system, which must guarantee transaction atomicity and durability in the face of system failures. We have developed a recovery mechanism based on non-interfering consistent checkpointing and log compression. On-line log compression is necessary to keep the log short to achieve a rapid restart. Compression can be used by any database system to improve restart time, but is essential for main memory database systems which may achieve very high transaction throughput. As opposed to most other recovery techniques, our technique has the advantage that a portion of memory can be made non-volatile by using batteries as a backup power supply. By exploiting this portion of non-volatile memory, log compression can be achieved effectively.

Performance of real-time database systems can be enhanced by the use of semantics of transactions and temporal data models, based on different notions of "correct execution" of transactions. Since all transactions are pre-compiled in many real-time database systems, semantic information such as types of transactions and data objects they need to access can be collected and used effectively. A read-only transaction is a typical example of the use of transaction semantics. A read-only transaction can be used to take a checkpoint of the database for recovering from subsequent failures, or to check the consistency of the database, or simply to retrieve the information from the database. Since read-only transactions are still transactions, they can be processed using the algorithms for arbitrary transactions. However, it is possible to use special processing algorithms for read-only transactions in order to improve efficiency, resulting in high performance.

Serializability has been accepted as the standard correctness criteria in database systems. However, people actually developing large real-time systems are unwilling to pay the price for serializability, because predictability of response is severely compromised due to blocking or preemption. For read-only transactions, correctness requirements can be divided into two independent classes: the currency requirement and the consistency requirement. The currency requirement specifies what update transactions should be reflected by the data read. The consistency requirement specifies the degree of consistency needed by read-only transactions: internal consistency, weak consistency, and strong consistency. High performance and reliability of the system can be achieved by using different correctness requirements for read-only transactions. Clearly, strong consistency is preferable in many situations to weak consistency. However, it can be cheaper to ensure weak consistency than to ensure strong consistency. For the applications that can tolerate a weaker requirement, the potential performance gain could be significant. We have investigated methods to specify correctness requirements and new techniques to combine them with data replication. Our research effort has resulted in a feasible solution using time-stamps and multiversions of data objects.

### 7.2. Development of A Database Prototyping Tool

One of the primary reasons for the difficulty in successfully developing and evaluating new techniques for distributed database systems is that it takes a long time to develop a system, and

evaluation is complicated because it involves a large number of system parameters that may change dynamically. Prototyping methods can be applied effectively to the evaluation of new techniques for implementing distributed database systems. By investigating design alternatives and performance/reliability characteristics of new database techniques, we can provide a clear understanding of design alternatives with their costs and benefits in quantitative measures. Furthermore, database technology can be implemented in a modular reusable form to enhance experimentation. Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation, especially for distributed real-time database systems.

A prototyping tool to implement database technology should be flexible and organized in a modular fashion to provide enhanced experimentation capability. A user should be able to specify system configurations such as the number of sites, network topology, the number and locations of processes, the number and locations of resources, and the interaction among processes. We use the client/server paradigm for process interaction in our prototyping tool. The system consists of a set of clients and servers, which are processes that cooperate for the purpose of transaction processing. Each server provides a service to the clients of the system, where a client can request a service by sending a request message to the corresponding server.

We have implemented a preliminary version of the prototyping tool running under StarLite on a Sun workstation. The current prototyping tool provides concurrent transaction execution facilities, including two-phase locking and timestamp ordering as underlying synchronization mechanisms. A series of experiments have been performed to test the correctness of the design and validity of the preliminary implementation of those two synchronization mechanisms. The primary performance metrics for the study were transaction response time, system throughput, and the number of aborted transactions. As a general rule, we found that transaction response time, in both mechanisms, increases with the increase of the degree of data distribution and the number of conflicts. The current prototyping tool also provides a multiversion data object control mechanism.

In a real-time database system, synchronization protocols must not only maintain the consistency constraints of the database but also satisfy the timing requirements of the transactions accessing the database. To satisfy both the consistency and real-time constraints, there is the need to integrate synchronization protocols with real-time priority scheduling protocols. A major source of problems in integrating the two protocols is the lack of coordination in the development of synchronization protocols and real-time priority scheduling protocols. Due to the effect of blocking in lock-based synchronization protocols, a direct application of a real-time scheduling algorithm to transactions may result in a condition known as *priority inversion*. Priority inversion is said to occur when a higher priority process is forced to wait for the execution of a lower priority process for an indefinite period of time. Priority inversion is inevitable in transaction systems. However, to achieve a high degree of schedulability in real-time applications, priority inversion must be minimized. We have been implementing priority-based scheduling algorithms in our prototyping environment, and investigating technical issues associated with them.

# Semantic Information and Consistency in Distributed Real-Time Systems

Sang Hyuk Son

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

## ABSTRACT

Considerable research effort has been devoted to the problem of developing techniques for achieving high availability of critical data in distributed real-time systems. One approach is to use replication. Replicated data is stored redundantly at multiple sites so that it can be used even if some of the copies are not available due to failures. This paper presents an algorithm for maintaining consistency and improving the performance of database with replicated data in distributed real-time systems. The semantic information of read-only transactions is exploited for improved efficiency, and a multiversion technique is used to increase the degree of concurrency. Related issues including version management and consistency of the states seen by transactions are discussed.

Index Terms: distributed system, replication, read-only transaction, consistency, multiversion.

## 1. Introduction

A distributed system consists of multiple autonomous computer systems (called *sites)* that are connected via a communication network. Since the physical separation of sites ensures the independent failure modes of sites and limits the propagation of errors throughout the system, distributed systems must be able to continue to operate correctly despite of component failures. However, as the size of a distributed system increases, so does the probability that one or more of its components will fail. Thus, distributed systems must be fault tolerant to component failures to achieve a desired level of reliability and availability. Asserting that the system will continue to operate correctly if less than a certain number of failure occurs is a guarantee independent of the reliability of the sites that make up the system. It is a measure of the fault tolerance supported by the system architecture, in contrast to fault tolerance achieved by using reliable components.

Considerable research effort has been devoted in recent years to the problem of developing techniques for achieving high availability of critical data in distributed systems. An obvious approach to improve availability is to keep replicated copies of such data at multiple sites so that the system can access the data even if some of the copies are not available due to failures. In addition to improved availability, replication can enhance performance by allowing user requests initiated at sites where the data are stored to be processed locally without incurring communication delays, and by distributing the workload of user requests to several sites where the subtasks of a user request can be processed concurrently. These benefits of replication must be seen in the light of the additional cost and complexities introduced by replication control.

A major restriction of using replication is that replicated copies must behave like a single copy, i.e., *mutual consistency* of a replicated data must be preserved. By mutual consistency, we mean that all copies converge to the same value and would be identical if all update activities cease. The inherent communication delay between sites that store and maintain copies of a replicated data makes it impossible to ensure that all copies are identical at all times when updates are processed in the system.

Mutual consistency is not the only constraint a distributed system must satisfy. In a system where several users concurrently access and update data, operations from different user requests may need to be interleaved and allowed to operate concurrently on data for higher throughput of the system. Concurrency control is the activity of coordinating concurrent accesses to the system in order to provide the effect that each request is executed in a serial fashion. The task of concurrency control in a distributed system is more complicated than that in a centralized system mainly because the information used to make scheduling decisions is itself distributed, and it must be managed properly to make correct decisions.

A number of concurrency control schemes proposed are based on the maintenance of multiple versions of data objects[BAY80, BER83, CHA85, REE83 SON86, SON87, STE81]. The objective of using multiple versions is to increase the degree of concurrency and to reduce the possibility of rejection of user requests by providing a succession of views of data objects. One of the reasons for rejecting a user request is that its operations cannot be serviced by the system. For example, a read operation has to be rejected if the value of data object it was supposed to read has already been overwritten by some other user request. Such rejections can be avoided by keeping old versions of each data object so that an appropriate old value can be given to a tardy read operation. In a system with multiple versions of data, each write operation on a data object produces a new version instead of overwriting it. Hence, for each read operation, the system selects an appropriate version to read, enjoying the flexibility in controlling the order of read and write operations.

A read-only transaction is a user request that does not modify the state of the database. A read-only transaction can be used to take a checkpoint of the database for recovering from subsequent failures, or to check the consistency of the database, or simply to retrieve the information from the database. Many applications of distributed databases for real-time systems can be characterized by a dominance of read-only transactions. Since read-only transactions are still transactions, they can be processed using the algorithms for arbitrary transactions. However, it is possible to use special processing algorithms for read-only transactions in order to improve efficiency, resulting in high performance. With this approach, the specialized transaction processing algorithm can take advantage of the semantic information that no data

will be modified by the transaction.

In this paper, we explore this idea of read-only transaction processing, and present a synchronization algorithm for read-only transactions in distributed environments. The algorithm is based on the idea of maintaining multiple versions of necessary data objects in the system, and requires read-only transactions to be identified to the system before they begin execution. By preventing interference between read-only transactions and other update transactions, the algorithm guarantees that read-only transactions will be successfully completed. In addition, the replication method used in the algorithm masks failures as long as one or more copies remain available.

There are several problems that must be solved by an algorithm that uses multiple versions. For example, selection of old versions for a given read-only transaction must ensure the consistency of the state seen by the transaction. In addition, the need to save old versions for read-only transactions introduces a storage management problem, i.e., methods to determine which version is no longer needed so that it can be discarded. In this paper, we focus our attention on these problems.

In the next section we present the basic concepts that are needed for this paper. Section 3 describes the execution of logical operations by corresponding physical operations. Section 4 describes our synchronization algorithm for replicated data. Section 5 presents two recovery procedures that can be used for replicated data objects, and Section 6 discusses the availability of replicated data. Section 7 concludes the paper.

## 2. Basic Concepts

A distributed database is a collection of data objects. Each data object has a name and is represented by a set of one or more replicated copies. Copies of a given data object should have the same value, although the values may be temporarily different due to update activities. In addition to data objects, a distributed database has a collection of *consistency constraints*. A consistency constraint is a predicate defined on the database which describes the relationships that must hold among the data objects and their values [ESW76].

Users interact with the database by submitting transactions. Each transaction represents a complete and correct computation, i.e., if a transaction is executed alone on an initially consistent database, it would terminate in a finite time and produce correct results, leaving the database consistent. A transaction consists of different types of operations such as read, write, and local computations. Read and write operations are used to access data objects, and local computations are used to determine the new values of data objects for write operations. Algorithms for replication control and synchronization pay no attention to local computations; they make scheduling decisions on the basis of the data objects a transaction reads and writes.

When a transaction commits, all the updates it made must be written permanently into the database. All participants must commit unanimously, implying that the updates performed by the transaction are made visible to other transactions in an "all or none" fashion. One of the most well-known techniques for the atomic commitment is a protocol called *two-phase commit* [SKE81], which works as the following:

In the first phase the coordinator sends "start transaction" messages to all the participants. Each participant individually votes either to commit the transaction by sending precommit message or to abort it by sending abort message, according to the result of the subtransaction it has executed. If a failure occurs during the first phase, consistency of the database is not violated, since none of the transaction's updates have yet been written into the database. In the second phase the coordinator collects all the votes and makes a decision. If all votes were precommit, the coordinator sends "commit" messages to the participants. If the coordinator had received one or more abort messages, it sends "abort" messages to the participants.

The standard correctness requirement for transactions is serializability. It means that the concurrent execution of a group of transactions is equivalent to some serial execution of the same group of transactions. For read-only transactions, correctness requirements can be divided into two independent classes: the currency requirement and the consistency requirement.

The currency requirement specifies what update transactions should be reflected by the data read. There are several ways in which the currency requirement can be specified; we are interested in the following two:

(1) Fixed-time requirement: A read-only transaction T requires data as they existed at a given time *t*. This means that the data read by the transaction must reflect the modifications of all update transactions committed in the system before *t*.

(2) Latest-time requirement: A read-only transaction T requires data it reads reflect at least all update transactions committed before T is started, i.e., T requires most up-to-date data available.

The consistency requirement specifies the degree of consistency needed by read-only transactions. A read-only transaction may have one of the following requirements:

(1) Internal consistency: It only requires that the values read by each read-only transaction satisfy the invariants (consistency constraints) of the database.

(2) Weak consistency: It requires that the values read by each read-only transaction be the result of a serial execution of some subset of the update transactions committed. Weak consistency is at least as strong a requirement as internal consistency, because the result of a serial execution of update transactions always satisfies consistency constraints.

(3) Strong consistency: It requires that all update transactions together with all other read-only transactions that require strong consistency, must be serializable as a group. Strong consistency requirement is equivalent to serializability requirement for processing of arbitrary transactions.

We make a few comments concerning the currency and consistency requirements. First, it might seem that the internal consistency requirement is too weak to be useful. However, a read-only transaction with only internal consistency requirement is very simple and efficient to process, and at least one proposed algorithm [FIS82] does not satisfy any stronger consistency requirement. Second, it is easy to see that strong consistency is a stronger requirement than weak consistency, as shown by the following example. Suppose we have two update transactions, $T_1$ and $T_2$, two read-only transactions, $T_3$ and $T_4$, and two

data objects, X and Y, stored at two sites A and B. Assume that the initial values of both X and Y were 0 before the execution of any transactions. Now consider the following execution sequence:

$T_3$ reads 0 from X at A.

$T_1$ writes 1 into X at A.

$T_4$ reads 1 from X at A.

$T_4$ reads 0 from Y at B.

$T_2$ writes 1 into Y at B.

$T_3$ reads 1 from Y at B.

The values read by $T_3$ are the result of a serial execution of $T_2<T_3<T_1$, while the values read by $T_4$ are the result of a serial execution of $T_1<T_4<T_2$. Both of them are valid serialization order, and thus, the execution is weakly consistent. However, there is no single serial execution of all four transactions, so the execution is not serializable. In other words, both read-only transactions see valid serialization orders of updates, but they see different orders.

Clearly, strong consistency is preferable to weak consistency. However, as in the case of internal consistency, it can be cheaper to ensure weak consistency than to ensure strong consistency. For the applications that can tolerate a weaker requirement, the potential performance gain could be significant.

Finally, one might wonder why fixed-time requirement is interesting, since most read-only transactions may require information about the latest database state. However, there are situations that the user is interested in looking at the database as it existed at a given time. For an example of a fixed-time read-only transaction, consider the case of a general in the army making a decision by looking at the database showing the current position of the enemy. The general may be interested in looking at the position of the enemy of few hours ago or few days ago, in order to figure out the purpose of their moving. A read-only transaction of a given fixed-time will provide the general with the desired results.

## 3. Execution of Logical Operations

In our algorithm, we use the notion of tokens to support a fault-tolerant distributed database in increasing both the availability of data and the degree of concurrency, without incurring too much storage and processing overhead. Each data object has a predetermined number of tokens. Tokens are used to designate a read-write copy, and a token copy is a single version representing the latest value of the data object. The site which has a token copy of a data object is called a *token site*, with respect to the data object.

Multiversions are stored and managed only at read-only copy sites. For read-only copies, each data object is a collection of consecutive versions. A read-only transaction does not necessarily read the latest committed version of a data object. The particular old version that a read-only transaction has to read is determined by the time-stamp of the read-only transaction (for the latest-time requirement) or by the given time (for the fixed-time requirement). The time-stamp is assigned to a read-only transaction when it begins, while the time-stamp for an update transaction is determined as it commits. When a read-only transaction with time-stamp T attempts to read a data object, the version of the data object with the largest time-stamp less than T is selected as the value to be returned by the read operation.

To simplify the presentation in this paper, we use a simple model of data objects, with only read and write operations, instead of considering an abstracted data model. As discussed in [HER86], greater concurrency among update transactions can be achieved if more semantic information about the specification of each abstract data object is used. The algorithm presented in this paper can be easily adapted to use this kind of semantic information of data objects.

In this paper, we do not consider Byzantine type of failures. When a site fails, it simply stops running (fail-stop). When the failed site recovers, the fact that it has failed is recognized, and a recovery procedure is initiated. We assume that site failures are detectable by other sites. This can be achieved either by network protocols or by high-level time-out mechanisms in the application layer.

We assume that update transactions use two-phase locking [ESW76], with exclusive locks used for write operations, and shared locks for read operations. Lock requests are made only to token copies, and there is no locks associated with read-only copies. In addition, update transactions use the two-phase commit protocol and stable storage [LAM81] to achieve fault-tolerance to site failures. When a new version is created, it is created at all copy sites, including read-only copy site. However, any new versions are not accessible to other transactions until they are finalized through the two-phase commit protocol. Upon receiving the commit message from the coordinator, new versions of data objects created by the transaction replace the current versions at token sites, while they are attached to the multiple versions at read-only sites.

Operations invoked by update transactions are processed using ordinary two-phase locking: when an update transaction invokes a read operation on a data object, it waits until it can lock the data object in shared mode. When an update transaction invokes a write operation, it locks the data object in exclusive mode, and then creates a new version. If the transaction later aborts, the newly created version will be discarded. Our algorithm follows the *read-one/write-all-available* paradigm [BHA86] in which a read lock request succeeds if at least one of the token copies can be locked in shared mode, and a write lock request fails if at least one of the available token copies cannot be locked in exclusive mode. In a straightforward implementation of a write operation in this paradigm, the value to be written is broadcast to all sites where a copy of the data object resides. A physical write operation occurs at each copy site, and then a confirmation message has to be returned to the site where the write operation was requested. The write operation is considered completed only when all the confirmation messages are returned. This solution is unsatisfactory because every write operation incurs waiting for responses before the next operation of the transaction can proceed. In the next section, we present an algorithm that permits an operation after a write to proceed as in a nonreplicated system, with the physical write operations being executed concurrently at other copy sites.

## 4. The Algorithm

As noted above, our algorithm combines time-stamp ordering and locking. To generate time-stamps for update transactions, a time-stamp is maintained for each data object. The time-stamp for data object X represents the maximum of the time-stamps of update transactions that have accessed X and committed, and the time-stamps of read-only transactions that have accessed X. Time-stamps for update transactions are generated during the commit phase as follows:

In the first phase of the two-phase commit protocol, each participant attaches to the precommit message the maximum time-stamps of all data objects that it accessed. Upon receiving precommit messages from the participants, the coordinator chooses a unique time-stamp greater than all the time-stamps received. This is the time-stamp for the transaction. Then, in the second phase of the commit protocol, this time-stamp is broadcast to all participants (piggybacked on the commit message). Each participant, upon receiving this message, updates the time-stamp of each data object to the maximum of its current value and the received time-stamp, and releases any locks held by the transaction. Any version written by the transaction is marked with the time-stamp of the transaction. Figure 1 shows the message passing between the coordinator and participants. Note that time-stamps are piggybacked on the precommit and commit messages, hence no additional messages are introduced here.

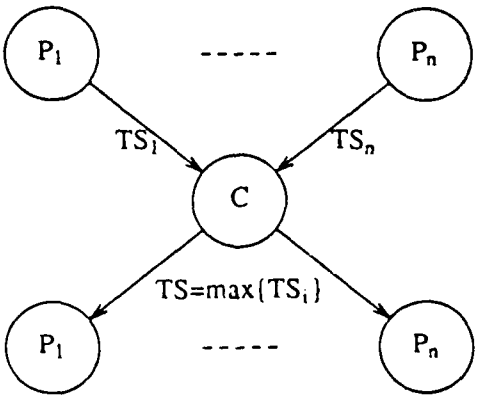



Fig. 1. Time-stamp generation for update transactions

When a participant searches for the maximum time-stamp to attach to a precommit message, read-only copies are also included in the set of copies for the search. By including the time-stamps of read-only copies in determining the time-stamp for an update transaction, the system can ensure that any potential conflicts between read-only transactions and the current update transaction are resolved in the correct order of their time-stamps.

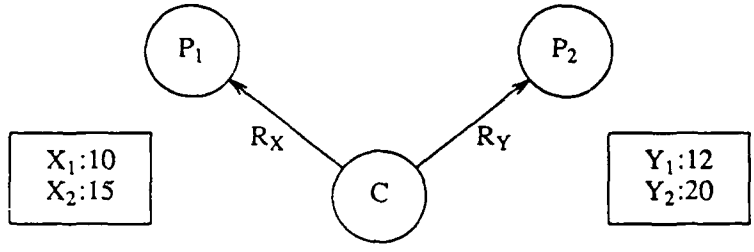
Time-stamp assignment for read-only transactions with latest-time requirement is quite different from that for update transactions. When a read-only transaction begins, the coordinator sends messages to the participants telling them the data objects the transaction needs to read. When a participant receives such a request, it checks the current time-stamp of each data object at the site, and sends the maximum time-stamps among them to the coordinator. Each data object accessed by a read-only transaction in this way records the pair of the identifier of that transaction and the current time-stamp it reported. After receiving responses from all participants, the coordinator chooses a unique time-stamp greater than all the responses. The time-stamp recorded for the read-only transaction at each object is thus a lower bound on the time-stamp of the transaction, and it will be used in making a decision to discard or retain versions of the data object. For a fixed-time read-only transactions, time-stamp is provided by the user, and hence the system needs not bother to assign a new time-stamp for it.

When a read-only transaction with time-stamp TS invokes a read operation on a data object, the participant chooses the version of the data object with the largest time-stamp less than TS. This invocation of read operation is nothing but sending the time-stamp TS to the participants, since each participant already knows which data object to read. If TS is larger than the current time-stamp of the data object, it will be updated as TS. This will force update transactions that commit later to choose time-stamps larger than TS, ensuring that the version selected for the read-only transaction does not change.

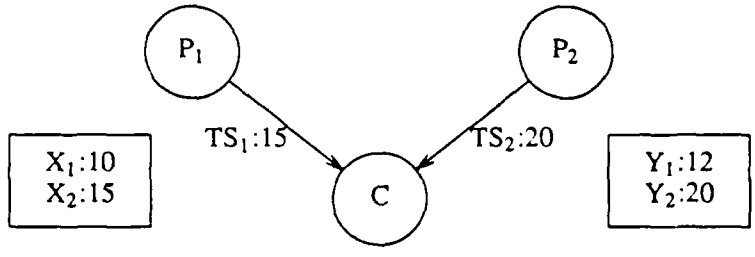
Figure 2 shows the operation sequence for time-stamp generation and read request processing for read-only transactions. The coordinator sends read requests for data object X and Y, each of which is maintained as two versions stored at participants $P_1$ and $P_2$. $P_1$ responds with time-stamp value of 15, and

$P_2$ with 20. The coordinator chooses a unique time-stamp (21 in this case), and sends it to each participant. Time-stamps of the data objects ($X_2$ and $Y_2$) are changed to 21 when read operation is completed.

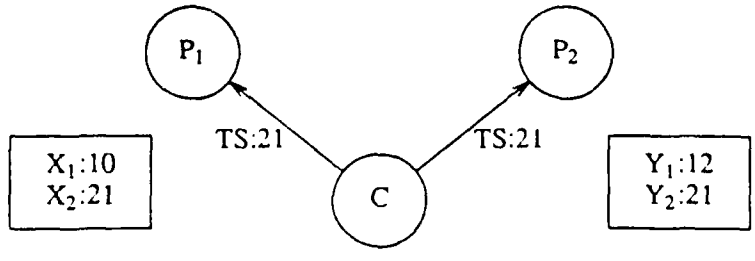
It is easy to show that the above algorithm ensures strong consistency. The mechanism for generating time-stamps for update transactions ensures that any conflicting update transactions are ordered according to their time-stamps, and hence they are serializable in the time-stamp order. Read-only




(a) read requests from the coordinator




(b) time-stamp report from participants




(c) read operation with unique time-stamp

Fig. 2. Time-stamp generation for read-only transactions

transactions then read versions of data objects consistent with all transactions executing in their time-stamp order.

To achieve the high performance by reducing the cost of write operations in our algorithm, the level of synchronization between write operation and its physical implementation can be relaxed by allowing physical write operations to be completed by the commit time of the transaction. A write operation is considered completed when the required update messages are sent. This eliminates the delay caused by waiting for confirmation messages before the next operation can proceed.

To this point we have assumed that all versions are retained forever. We now discuss how versions can be discarded when they are not needed by read-only transactions. Recall that each data object keeps track of the read-only transactions that have accessed the data object, along with a lower bound on the time-stamp chosen by each transaction. Data objects can use the following rule to decide which versions to keep and which to discard.

Rule for retention:

A version with time-stamp TS must be retained if

(1) there is no version with time-stamp greater than TS (i.e., current version), or

(2) there is a version with time-stamp TS' > TS, and there is an active read-only transaction whose time-stamp might be between TS and TS'.

By having a read-only transaction inform data objects when it completes, versions of data objects that are no longer needed can be discarded. This process of informing data objects that a read-only transaction has completed need not be performed synchronously with the commit of the transaction. It imposes some overhead on the system, but the overhead can be reduced by piggybacking information on existing messages, or by sending messages when the system load is low.

When a read-only transaction sends a read request to an object, the read-only site effectively agrees to retain the current version and any later versions, until it knows which of those versions is needed by the

read-only transaction. When the read-only site finds out the time-stamp chosen by the transaction, it can tell exactly which version the transaction needs to read. At that point any versions that were retained only because the read-only transaction might have needed them can be discarded. By minimizing the time during which only a lower bound on the transaction's time-stamp is known, the system can reduce the storage needed for maintaining versions. One simple way of doing this is to have each read-only transaction broadcast its time-stamp to all read-only sites when it chooses the time-stamp.

The version management described above is effective at minimizing the amount of storage needed for versions. For example, unlike the "version pool" scheme in [CHA85], it is not necessary to discard a version that is needed by an active read-only transaction because the buffer space is being used by a version that no transaction wants to read. However, ensuring that each read-only site knows which versions are needed at any point in time has an associated cost; a read-only transaction cannot begin execution until it has chosen a time-stamp, a process that requires communicating with all data objects it needs to access.

Because the time-stamp for a fixed-time read-only transaction is determined by the user, the number of versions that needs to be retained to process fixed-time read-only transactions cannot be bounded as in the case for latest-time read-only transactions. In order to process all the potential fixed-time read-only transactions, the system must maintain all the versions created up to the present, which may require huge amount of storage. There are several alternatives to keep a history instead of saving all the versions created for each data object. One of the simplest and efficient alternative would be to keep a log of all the update transactions. A transaction log is a record of all the transactions and the updates they performed. Fixed-time read-only transactions can be processed by examining the log in reverse chronological order until the desired version of the data object can be reconstructed. Since fixed-time read-only transactions must examine the log, their execution depends on the availability of the log, and their execution speed would be slower than that of latest-time read-only transactions. One important advantage of the transaction log mechanism is that in many systems the log is required anyway for crash recovery. Thus, in these systems, keeping the log for fixed-time read-only transactions represents no real overhead.

Performance of synchronization algorithms can be evaluated by several aspects: storage requirement, number of aborts, and average transaction response time. Storage requirement is proportional to the number of versions to be maintained in the system. The algorithm presented in this paper achieves storage reduction by two methods: token copies and version retention rule. The amount of reduced storage requirement by introducing token copies is a function of the number of tokens for data objects and the average number of versions to be maintained for each data object. For example, if two copies are token copies for a four-copy data object with ten versions, we can save twenty versions for that data object. Total reduction of storage requirement for the database is

$$\text{Storage reduction} = \sum_{i=1}^{N} (\text{number of token copies of data}_i) \times (\text{number of versions of data}_i)$$

where N is the number of data objects in the database.

Version retention rule also contributes to the reduction of the number of versions by allowing the system to maintain only those versions that will be used for read operations.

Read-only transactions are never aborted and their response time is reduced because they do not need to go through two-phase commit protocol. Furthermore, access requests from read-only transactions do not require to access token copies, and hence no blocking is introduced by update transactions. This results in further reduction of response time of read-only transactions. In general, the number of aborts and average transaction response time for a given set of transactions depend on system parameters and read-set/write-set of transactions, making analytical evaluation complicated. A prototyping tool for experimenting distributed database systems is being developed at the University of Virginia, and a quantitative evaluation of the proposed algorithm will be performed and reported in a separate paper.

## 5. Concluding Remarks

Replication is the key factor in making distributed systems more reliable than centralized systems. However, if replication is used without proper synchronization mechanisms, consistency of the system might be violated. In this paper, we have presented a synchronization algorithm for distributed real-time

systems with replicated data. It reduces the time required to execute physical write operations when updates are to be made on replicated data objects, by relaxing the level of synchronization between write operations on data objects and physical write operations on copies of them. At the same time, the consistency of replicated data is not violated, and the atomicity of transactions is maintained. The algorithm exploits the multiple versions of a data object and the semantic information of read-only transactions in achieving improved system performance. The algorithm also extends the notion of primary copies such that an update transaction can be executed provided at least one token copy of each data object in the write set is available. The number of tokens for each data object can be used as a tuning parameter to adjust the robustness of the system. Multiple versions are maintained only at the read-only copy sites, hence the storage requirement is reduced in comparison to other multiversion mechanisms[REE83, CHA85].

Reliability does not come for free. There is a cost associated with the replication of data: storage requirement and complicated control in synchronization. For appropriate management of multiple versions, some communication cost is inevitable to inform data objects about activities of read-only transactions. There is also a cost associated with maintaining the data structures for keeping track of versions and time-stamps. In many real-time applications of distributed databases, however, the cost of replication is justifiable. Further work is clearly needed to develop alternative approaches for maintaining multiversions and exploiting semantic information of read-only transactions, and to study performance of different approaches.

## ACKNOWLEDGEMENTS

This work was supported in part by the Office of Naval Research under contract no. N00014-86-K-0245, and by the Jet Propulsion Laboratory of the California Institute of Technology under contract no. 957721 to the University of Virginia.

## REFERENCES


BAY80 Bayer, R., Heller, H., and Reiser, A., Parallelism and Recovery in Database Systems, ACM Trans. on Database Systems, June 1980, pp 139-156.

BER83 Bernstein, P., Goodman N., Multiversion Concurrency Control - Theory and Algorithms, ACM Trans. on Database Systems, Dec. 1983, pp 465-483.

BHA86 Bhargava, B., Ruan, Z., Site Recovery in Replicated Distributed Database Systems, Proc. 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 1986, pp 621-627..

CHA85 Chan, A., Gray, R., Implementing Distributed Read-Only Transactions, IEEE Trans. on Software Engineering, Feb. 1985, pp 205-212.

ESW76 Eswaran, K.P. et al, The Notion of Consistency and Predicate Locks in a Database System, CACM 19, Nov. 1976, pp 624-633.

FIS82 Fischer, M. J., Griffeth, N. D. and Lynch, N. A., Global States of a Distributed System, IEEE Trans. on Software Engineering, May 1982, pp 198-202.

GOO83 Goodman, N., Skeen, D. and et al., A Recovery Algorithm for a Distributed Database System, Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 1983, pp 8-15.

HAM80 Hammer, M. and Shipman, D., Reliability Mechanisms for SDD-1: A System for Distributed Databases, ACM Trans. on Database Systems, December 1980, pp 431-466.

HER86 Herlihy, M., A Quorum-Consensus Replication Method for Abstract Data Types, ACM Trans. on Computer Systems, February 1986, pp 32-53.

LAM81 Lampson, B., Atomic Transactions, Distributed Systems: Architecture and Implementation, Lecture Notes in Computer Science, Vol. 105, Springer-Verlag, 1981, pp 246-265.

REE83 Reed, D., Implementing Atomic Actions on Decentralized Data, ACM Trans. on Computer Systems, Feb. 1983, pp 3-23.

SKE81 Skeen, D., Nonblocking Commit Protocols, Proc. ACM SIGMOD Conference on Management of Data, 1981, pp 133-142.

SKE85 Skeen, D., Determining The Last Process to Fail, ACM Trans. on Computer Systems, Feb. 1985, pp 15-30.

SON86 Son, S. H. and Agrawala, A., A Token-Based Resiliency Control Scheme in Replicated Database Systems, Proc. Fifth Symposium on Reliability in Distributed Software and Database Systems, January 1986, pp 199-206.

SON86b Son, S. H. and Agrawala, A., An Algorithm for Database Reconstruction in Distributed Environments, Proc. 6th International Conference on Distributed Computing Systems, May 1986, pp 532-539.

SON87 Son, S. H., On Multiversion Replication Control in Distributed Systems, Computer Systems Science and Engineering, Vol. 2, No. 2, April 1987, pp 76-84.

STE81 Stearns R. E., Rosenkrantz, D. J., Distributed Database Concurrency Controls Using Before-Values, Proc. ACM SIGMOD Conference, 1981, pp 74-83.

# The StarLite Prototyping Architecture

## 1. Introduction

The goal of the StarLite project is to test the hypothesis that a host prototyping environment can be used to significantly accelerate our ability to perform experiments in the areas of operating systems, databases, and network protocols. This paper discusses the requirements for an architecture to support software prototyping and the current StarLite implementation. The requirements suggest an architecture quite different from the RISC architectures currently in vogue. However, the resulting interpreter has characteristics that make it ideally suited to execute on current RISC machines.

The primary project requirement for StarLite is that software developed in the prototyping environment must be capable of being retargeted to different architectures only by recompiling and replacing a few low-level modules. The anticipated benefits are fast prototyping times, greater sharing of software in the research community, and the ability for one research group to validate the claims of another by replicating experimental conditions exactly.

The components of the StarLite project include a Modula-2 compiler, a symbolic debugger, an interpreter for the prototyping architecture, and a visual simulation package. The compiler and interpreter are implemented in C for portability; the rest of the software is in Modula-2. The prototyping environment has been used to develop a non-proprietary, UNIX-like operating system that is designed for a multiprocessor architecture, as well as to perform experiments with concurrency control algorithms for distributed database systems.

As one measure of the effectiveness of the environment, it is often possible to fix errors in the operating system, compile, and reboot the StarLite virtual machine in less than twenty seconds. The compilation time on a SUN 3/280 for the 66 modules (7500 lines) that comprise the operating system is one minute (clock) or 16 seconds (user) time. The StarLite VM, as measured by Wirth's Modula-2 benchmark program[1], executes at a speed of from one to six times that of a PDP 11/40, depending on the mix of instructions.

## 2. Architectural Requirements for Prototyping

The StarLite prototyping architecture is designed to support the simultaneous execution of multiple operating systems in a single address space. For example, to prototype a distributed operating system, we might want to initiate a file server and several clients. Each virtual machine would have its own operating system and user processes. All of the code and data for all of the virtual machines would be executed as a single UNIX process.

In order to support this requirement, we assume the existence of high-performance workstations with large local memories. Ideally, we would prefer multi-thread support, but multiprocessor workstations are not yet widely available. We also assume that hardware details can be isolated behind high-level language interfaces to the extent that the majority of a system's software remains invariant when retargeted from the host to a target architecture.

The architectural requirements to be satisfied by an interpreter that supports multiple operating systems running in a single, large address space are interesting. They include high speed, compact code, exception handling, good error detection, demand loading, dynamic restart, fast context switches, hybrid execution modes, and portability.

In the following sections, we will justify our requirements.

**High Speed.** Obviously, the speed of the host architecture is a determining factor in the usefulness of any prototyping effort. Prototyping is most effective for logic-intensive programs, such as operating systems, because the ratio of code to code-executed-per-function is high. For example, running user programs at the shell level on top of the prototype operating system, which is running on an interpreter, provides a response-level comparable (several seconds ) to a PDP-11. As the number of users increase or as the number of data-intensive applications increase, the response time increases considerably. Data-intensive programs tend to apply a large percentage of their code to each data point. Thus, the number of data points determines execution speed. In many cases, having fast machines is the only effective way to prototype data-intensive applications.

Since the StarLite system uses an interpreter to define its virtual machines, we tend to stay away from data-intensive test programs. It would be nice to have an execution speed comparable to a bare machine, but that could only be achieved by building a software prototyping workstation. For now, we are satisfied as long as the edit-compile-boot-and-test cycle is significantly faster than any other environment.

The interpreter is implemented as a single procedure to take advantage of C's *register* declaration. For example, an earlier version of the interpreter used static variables for the registers. The conversion to *register* variables improved performance by a factor of three on a SUN 3/280.

Since the SUN 3/280 is based on the M68020, the number of virtual machine registers that could be assigned to hardware registers was limited to three. Above that number, performance started to drop off as the C compiler generated extra code to compensate for the reduction in usable registers. In current RISC machines, such as MIPS or the SUN 4, this limit would not be a problem. The use of multiple registers, and the lack of context switches, as well as a minimal number of procedure calls should enable the interpreter to take full advantage of the characteristics of these processors.

The StarLite architecture is a 32-bit extension of Wirth's Lilith architecture[1], which in turn is a descendent of the Xerox Alto processor. It is also a contraction. For instance, the Lilith uses an evaluation stack of registers with a hardware stack pointer and no overflow/underflow checking. This is a good idea in a hardware implementation; however for the prototyping architecture, it would result in memory-to-memory copies of the evaluation stack on procedure calls and context switches. Therefore, we modified the architecture to support a pure stack model of execution.

The instruction stream is byte-coded and there are no highly encoded instructions. As a result the physical interpretation is fast; for example, the interpreter's main loop is 5 instructions on a M68020. This could be reduced to two instructions by generating threaded code but that would negate the portability goal.

With an interpreter, the instruction set architecture can assign lots of functionality to each instruction. This has two advantages. First, the function is executed in hardware, which makes it fast. Second, the overhead of multiple passes through the interpretation loop is saved. The instruction set has been carefully tuned by analyzing all of the system code for the Lilith environment and by using the Xerox analysis[2] of Mesa.

The StarLite environment actually supports a family of architectures for which the interpreters have different characteristics. For example, adding a single-step trace option costs three additional instructions per loop. In another version, the virtual clock is driven by instruction execution (1 tick per 100 instructions). Having the clock regulated by instruction execution is advantageous for optimizing code and experimenting with real-time systems; however, it costs an additional three instructions per loop.

Another version of the architecture was created just to support the UNIX "fork" operation, which requires dynamic relocation on data references. This feature does not slow the interpretation loop but rather it has an effect on every instruction that loads or stores data. It is possible to continue increasing the degree of detail in the interpreter until a level equivalent to IBM's VM emulation[3] is achieved. However, this results in a significant decrease in execution speed. It also has the further disadvantage of focusing the programmer's attention on hardware details at the expense of further refinements in system abstractions.

**Compact Code.** The generated code for the StarLite architecture is extremely space efficient since it is based on the Lilith design. For example, the object code (.o file) sizes for a sample 1,000 line program were SUN3-Modula2(130K), SUN3-C(65K), PC286-C(35K), StarLite-Modula2(11K). Compact code has a significant effect on the speed with which the environment can load both system components and user-level programs that might run on those components. Compactness also increases cache locality, reduces page faults, and maximizes the quantity of software that can be co-resident in the prototyping system.

**Exception Handling.** The benefits of exception handling support for large system development have already been documented by Rovner[4].

**Error Detection.** The benefits of integrity checking for an architecture's primitive operations have been discussed by Wirth[5]. The StarLite architecture supports checks for overflow/underflow, division by zero, subrange and subscript checking, NIL pointer checks, illegal addresses, and stack overflow. Subrange and subscript checks are generated by the compiler. The other faults should be detected by the underlying C runtime. If not, we use the C compiler's "-a" option and then modify the assembler output with an editing script.

**Demand Loading.** The StarLite architecture supports demand loading; that is, modules are loaded at the point that one of their procedures is called. Thus, a large software system begins execution very quickly and then loads only the modules that are actually referenced. For example, one version of the operating system defers loading the file system, or even the disk driver, until a file operation is performed.

Achieving this requirement was complicated by the format of module initialization code. In Modula-2, the initialization procedures for all imported modules must be invoked prior to executing the initialization code for the importing module. Without some care in the architecture definition, all modules would be demand loaded as soon as a program module began execution, which would negate the benefits. The solution is described in Section 3.1.4.

At the current time, a linker is superfluous; as soon as a module is compiled, it may be executed. Demand loading and the absence of linking greatly enhances the efficacy of the StarLite debug cycle. The only limit on debugging is how fast the programmer can discover bugs and type in the changes.

**Dynamic Restart.** When debugging software, it can be annoying to discover an error, return to the host level, compile, and then run the system to the point of error only to discover another silly mistake. The StarLite architecture is designed so that an IMPLEMENTATION module can be compiled in a child process while the interpreter is suspended. That module can be reinserted into memory and the system restarted.

Another dynamic restart feature supports the emulation of partial failure as might be experienced in a distributed system. The Modula-2 compiler does not attempt to statically initialize any data area. Thus, any module, or set of modules, can be dynamically restarted at any

time without reloading the object modules from disk. For a distributed system, the user can induce virtual processor failures and then "bring up" the operating system on those nodes without loading any software from disk.

**Fast Context Switches.** Unlike the "high-speed" requirement, achieving a fast context switch time can be realized independent of the characteristics of the host machine. For example, there are no context switches within the interpreter, which is basically a C procedure in a closed loop. Therefore, a host architecture with a slow context switch time has no effect on the interpreter's context switch time; it is only a function of the state information that must be saved and restored. This is an important requirement as a typical operating system "run" can involve thousands of context switches.

The StarLite architecture specifies only a single register (P) per thread. The additional state information (described later) is located in the thread's stack. Whether or not the additional state information is implemented as registers or is left on the stack is an implementation decision. For example, a machine with a well-matched caching strategy could support an implementation that left the state on the stack and used P as a base register. The result would be a context switch that involved only changing P registers. On the other hand, if the state information were copied into C *register* variables to improve performance, a context switch would involve register save/restore operations. Each implementation of the architecture must be balanced to match the characteristics of the host machine. The current SUN 3/280 interpreter executes 200,000 coroutine transfers/second. On the other hand, the IBM PS2/50 interpreter executes at 10,000 transfers/second.

**Hybrid Execution Modes.** In a prototyping environment, it is advantageous to use services that already exist in the host environment. For example, it is possible to "mount" the host file system on a leaf of a prototyped file system, or even as the prototype's "root" file system. Another example would be to use the host's database services.

Yet another example occurs in situations where the prototype would execute partially in the host and partially in a target system. An illustration of this case would be the use of a physical disk server by an operating system running in the host prototyping environment.

The keys to hybrid execution are architectural support and the definition of interfaces that remain invariant to changes in implementation technology. For example, the following interface is used in the operating system.

```
PROCEDURE Load(VAR programName : ARRAY OF CHAR):BOOLEAN;
```

It is used by the "exec" system call to load user programs into memory. The interface "hides" implementation details such as the existence of a prototype file system or the virtual memory architecture. This "information hiding" principle is also used in designing device interfaces. As a result, the operating system never knows whether devices, such as disks, or services, such as "Load", are real or are emulated.

In the case of "Load", for example, it is included in the architecture's collection of VM ROM routines. If the user does not supply a "Loader" module, the one in VM ROM will be used instead. A VM ROM routine has a DEFINITION module but its implementation is part of the interpreter. VM ROM can be used to provide functionality that the prototype software does not. For example, when prototyping an operating system to experiment with file system issues, it is not necessary to worry about program management; VM ROM routines can be used to take care of the details.

It is easy to add additional packages to the VM ROM interface. The disadvantage is that all ROM packages, which are written in C by the way, must be co-resident with the interpreter. In a future version of StarLite under IBM's OS/2, all of the ROM packages will be dynamically linked on demand.

**Portability.** One of the benefits of developing systems in the StarLite environment is that the code can be shared with other researchers. To facilitate sharing at the object code level, the instructions generated by the compiler and its object module format are canonical. That is, the byte ordering is fixed, as is the character code (ASCII), and the floating point format (IEEE). If the host has different conventions, the compiler performs the conversions as it generates code. To the extent that an implementation module is machine invariant, it should be possible to transmit object modules from one site to another and to have them work.

The StarLite operating system design project is experimenting with the use of "safe"[4], canonical object modules for user-specified line and protocol filters, schedulers, and application-specific file systems. For example, the operating system stores method descriptions for file access in the canonical object code format. The advantage of a canonical representation is that the volume can be transported to a different machine, which could then interpret the access method to manipulate the volume.

## 3. The StarLite Architecture

The StarLite architecture extends the Lilith design to satisfy the requirements for prototyping. It supports the INTEGER, CARDINAL, LONGINT, REAL, PROCEDURE, and BITSET types and includes modifications to the coroutine structure. This section first describes the components of a coroutine, which is the fundamental building block for emulating various processor configurations. Secondly, the instruction set architecture is explained. Following Wirth[1], the definitions are in Modula-2.

### 3.1 The coroutine structure

A coroutine, or thread, in the StarLite architecture consists of state information, code, global data, and a stack. Coincidentally, this is the model supported by most implementations of UNIX. A coroutine is defined as follows:

```
TYPE
Coroutine =
   xState : ExecutionState;    (* where execution is/was *)
   interruptMask : BITSET;        (* 1 is enabled        *)
   interruptVectorPointer : pInterrupt;
   topOfStackPointer, stackLimitPointer : pStack;
   dataFrameTablePointer : pDataFrameTable;   (* one base register/module to locate global data *)
   moduleInfoTablePointer : pModuleInfoTable; (* describes loaded modules *)
   eventInfo : EventDescription;  (* reason for the latest internal/external trap *)
   bottomFrame : Frame;           (* describes the base of the procedure activation record stack *)
END; (* Coroutine *)

Stack  = ARRAY [0..MAXSTACK] OF WORD;  (* generic area in a coroutine's stack *)
pStack = POINTER TO Stack;
```

The coroutine structure is designed for fast context switching. A context switch can be accomplished by exchanging two coroutine pointers. Unfortunately, leaving all the state information in a coroutine's stack can significantly slow execution on some architectures. Therefore, some experimentation is necessary each time the interpreter is ported to determine a good balance between context-switch time (all registers in the coroutine record) and execution speed (some/all registers in C *register* variables).

The standard portion (relative to most other architectures) of the state information in the coroutine record contains the current instruction position (xState), top-of-stack and stack limit pointers, an interrupt mask, and a "bottom" frame that "marks" the base of the procedure activation record stack.

The non-standard components of the coroutine record are the event descriptor (eventInfo) and the pointers to the interrupt vector, data frame and module information tables.

### 3.1.1 The interrupt vector

By using an interrupt vector pointer, all coroutines can share the same trap/interrupt vector or the coroutines can be partitioned to use different vectors. As a result, it is possible to emulate a multiprocessor or distributed processor architecture by changing the InitCoroutine procedure. This can be accomplished without modifying the interpreter. It also means that the interpreter is unaware and unaffected by the number of virtual processors that higher-level software creates.

The interrupt vector, which also handles exceptions, is implemented as an array of pointers to coroutines. An interrupt, then, is just a Modula-2 "transfer" operation on two coroutines, the one executing and the one identified by the appropriate vector entry. At the present time, entry zero is for the system clock interrupt and entry one is for program faults. Other interrupt options that we considered (but rejected) were procedure variables and semaphores. Both choices had disadvantages that the use of coroutines avoided.

With respect to exceptions and interrupts, there is a somewhat symbiotic relationship between the interpreter and the operating systems that live on top of it. For example, a procedure return from the "bottom" frame raises the "normal exit" exception.

If the interrupt vector entry for this exception is NIL, the interpreter terminates the "booted" program. In a bare machine, there is nothing "underneath" so a NIL vector location induces a machine fault. However, the StarLite VM is not "bare" so it implements reasonable actions for what would otherwise be unrecoverable errors in hardware. At the VM level above the interpreter, the prototype operating system handles the "normal exit" exception by executing an "Exit(0)" system call on behalf of the process.

```
TYPE
Interrupt = ARRAY [0..MAXINTS] OF pCoroutine;  (* Trap/interrupt vector *)
pInterrupt = POINTER TO Interrupt;                (* part of state information *)

EventDescription = RECORD                (* records why a process is handling an exception/event *)
   eventCode : CARDINAL;  (* Possible values - normal, halted, caseerr, stackovf, heapovf, functionerr,
                addresserr, realovf, realunf, badoperand, cardinalovf, integerovf, rangeerr,
                dividebyzero, illegalinst, breakpnt, singlestep, missingmod *)
   eventString : ARRAY [0..MAXMSG] OF CHAR;
END; (* EventDescription *)
```

### 3.1.2 Event processing

The "eventInfo" field in the StarLite coroutine record provides integrated support for exception handling in the architecture. Other Modula-2 systems[4] have modified the language to support exception handling; we provide similar functionality, but not the syntax, via an Exception module implemented in Modula-2. We use the terms "catch" to refer to the establishment of a handler and "raise" to refer to the action when an exception is detected. The modifications to the Lilith architecture involved adding an exception handler pointer cell to each activation record, adding a RESTORE instruction to restore the context for a handler, and adding the event fields to the coroutine record.

The StarLite architecture is somewhat unique in its handling of exceptions, or events at the architectural level. First, an exception may be raised in three ways. It may occur through program execution, such as for division-by-zero, through the execution of the Exception.Raise procedure, or through the intervention of another coroutine. The first two methods are traditional; the latter is not.

There are two reasons for providing support in the architecture for inter-coroutine exceptions. First, some exceptions, such as stack overflow, may not be appropriate for the executing coroutine to "catch". If it has corrupted its stack, it may not continue execution safely. Every coroutine has a "more trustworthy" coroutine to "catch" its exceptions. The "most trustworthy" catcher is the interpreter, which intervenes for NIL vector entries.

The second reason is that inter-coroutine exceptions are a good way to "back" a coroutine out of a module hierarchy where it holds locks or other resources. For instance, the operating system's "kill" implementation could use this technique by sending a "kill" exception to an operating system thread executing a system call on behalf of a user. When the thread "catches" the exception, it must exit each module that it was executing and release any allocated resources.

The problem with inter-coroutine exceptions is what to do when multiple coroutines perform a "raise" operation on the same victim. In the StarLite architecture, when the eventCode is set to non-zero, an exception is raised atomically with respect to any other exception for a given coroutine. When a handler sets the cell back to zero, the atomic action has completed and other exceptions can be raised. If a coroutine generates or attempts to raise an exception while in this mode, it is killed. The rationale is that the handler is supposed to be "more trusted"; if it fails, the program is in trouble.

The eventString field was introduced to reduce the need for a plethora of routines to invert error codes to sensible messages. It is intended to be used for the error message that matches the eventCode; however, the field can also be used for path names of error message files or for data.

### 3.1.3 Module storage

The StarLite architecture, which is based on Wirth's Lilith architecture, is designed to manage the storage for modules. The prototyping environment is designed to support debugging modules of code. Data storage is not yet an issue since we assume that the host workstation's virtual, if not physical, memory is sufficient to store a system while it is being prototyped.

The data frame table (DFT) is the focal point of all inter-module addressing, both for data and code. The DFT is indexed by module number. All variables and procedures external to a module are addressed via a module-number/offset pair. For variables, the offset is the data address. For procedures, the offset is the procedure number within its module. Loading a module, then, involves placing its code and data in memory and replacing the references to external module numbers with their DFT indices.

The data frame table (DFT) has one entry for every linked module. As with interrupt vectors, there can be as many DFTs as needed to emulate multiprocessor or distributed architectures. Each DFT entry contains either the pointer to the data structure for a loaded module, a marker that indicates a linked, but not loaded, module, a marker that indicates an unused entry, or an "extraCode" marker.

The "extraCode" marker identifies a routine, such as InOut, that is stored in VM ROM. References to an "extraCode" module are intercepted by the interpreter and are directed to C routines. The ROM feature supports the creation of prototypes that execute in hybrid mode, some parts are in VM and some parts are in VM ROM. Any function available to a C program on the host operating system is available by the use of the VM ROM technique.

The following definition describes the storage structure for a loaded module. Each DFT entry contains the base address of the global data area for a module. As mentioned earlier, the only static data generated by the compiler are string constants, which are located by a pointer cell. The initialization flag is set by a test-and-set instruction when a module's initialization code is executed. The setting of the flag indicates that the initialization code is being executed. To dynamically restart a module, it is only necessary to reset this flag and the module will repeat its initialization sequence. The code pointer locates the code vector associated with a module. The code vector contains a vector of offsets to the code bytes for each of a module's procedures, followed by the code bytes.

```
TYPE
pDataFrameTable = POINTER TO DataFrameTable;        (* part of the state information *)
DataFrameTable = ARRAY [0..MAXMODS] OF pModule; (* identifies the code/data for loaded modules *)

pModule = POINTER TO Module;
Module  = RECORD
   codePointer : pCode;              (* locates the code for a module *)
   initializedFlag : LONGINT;        (* set to non-zero when module initialization begins *)
   stringPointer : pStringArea;      (* pointer to the static data area where string constants are stored *)
   globalData : ARRAY [0..MAXGLOBALS] OF WORD;   (* global data (includes string constants) *)
END; (* Module *)

pStringArea = POINTER TO StringArea;
StringArea  = ARRAY [0..MAXPROGRAMSTRINGS] OF CHAR; (* constants generated by compiler *)


pCode = POINTER TO Code;
TwoViews = (OffsetView, CodeByteView);
Code = RECORD
   CASE :TwoViews OF
    OffsetView:
     codeOffsets : ARRAY [0..MAXPROCS] OF CARDINAL;  (* offsets to the code for each procedure *)
    |
     codeBytes : ARRAY [0..MAXCODEBYTES] OF Opcodes;  (* instructions for a module's procedures *)
   END; (* case *)
END; (* Code *)

pModuleInfoTable = POINTER TO ModuleInfoTable;
ModuleInfoTable  = RECORD                    (* identifies all referenced modules *)
   name : ARRAY [0..MAXNAME] OF CHAR;        (* the module's name *)
   key  : ARRAY [0..MAXKEY] OF CARDINAL;     (* timestamp of the corresponding DEFINITION part *)
   modNo: [0..MAXMODS];                      (* identifies the DFT entry assigned to the module *)
   codeSize : CARDINAL;                      (* the size of Code *)
   dataSize : LONGINT;                       (* the size of Module *)
END; (* ModuleInfoTable *)
```

The module information table (MIT) is defined as part of the architecture in an effort to keep the debugger's code independent of the structure of the software being prototyped. The table is also used by the routine that implements demand loading. As described earlier, if the vector entry for the "missingmod" exception is NIL, the interpreter invokes an extracode routine to load the module with a "name" and "key" that matches the module referenced by the instruction. The module number from the instruction can be inverted to an MIT entry by a linear search.

### 3.1.4 Procedure activation records

There are four classes of procedure activation records: coroutine start, local call, inter-module call, and dynamic initialization call. Coroutine-start is used as a marker for the bottom

frame when InitCoroutine creates a coroutine. The local-call case contains the standard static and dynamic links and a saved PC value. For an inter-module call, the module number of the caller is saved. When a procedure return occurs, the G value is sufficient to retrieve the base addresses of both the global data area and the code vector.

Finally, the dynamic-initialization case, which is new to StarLite, is used for demand loading. The initialization code for each module contains a call to the initialization procedure (0) for every imported module. For demand loading to work, these calls are treated as NOPs. Otherwise, all modules would be loaded as soon as the program module executed its initialization code.

As a result, when an external variable or procedure (other than zero) is referenced and the DFT entry is marked as loadable, the module must be loaded and initialized. The initialization is initiated by "faking" a procedure call at the point of reference. The dynamic-initialization marker indicates this special case. Also, since the "missingmod" exception leaves the PC at the instruction that generated the fault, it will be restarted when the initialization completes.

```
TYPE
Frame = RECORD                    (* format for a procedure's activation record *)
   xState : ExecutionState;       (* where execution was *)
   handler: ADDRESS;              (* address of the exception handler for this frame *)
   localData : Stack;             (* space for a procedure's local variables *)
END; (* Frame *)

StateViews = (CoroutineStart, InterModuleCall, DynamicInitialization, LocalCall);

ExecutionState = RECORD           (* used to start/restart execution and to return from a procedure *)
   CASE :StateViews OF
    CoroutineStart:
    |
    InterModuleCall, DynamicInitialization:
     G : [0..MAXMODS+256];        (* number of calling module; G&100 => dynamic initialization *)
    |
    LocalCall:
     staticLink : POINTER TO Frame;  (* L register (statically) of the calling procedure *)
   END; (* case *)
   L : POINTER TO Frame;          (* back link to the frame for the caller; dynamic link *)
   PC: CARDINAL;                  (* PC=FFFF => the bottom frame; PC&8000 => marks an intermodule call *)
END; (* ExecutionState *)
```

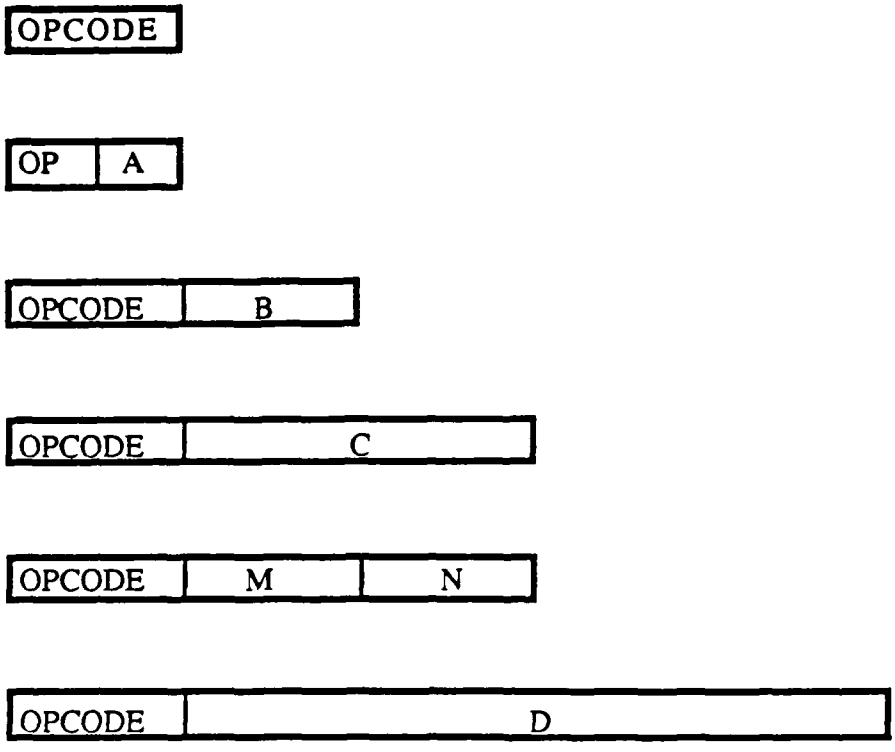
## 4. The Instruction Set Architecture

In this section, we outline the StarLite extensions to the Lilith architecture. For those interested in more detail, Appendix A presents a system consisting of two modules, together with a decoding of the object module for each. Furthermore, there is a memory snapshot that illustrates the "main" coroutine's data structures. The snapshot shows the program state at load time, during dynamic initialization following a demand load, and during a nested procedure call.

The instruction format of the StarLite architecture is identical to that of the Lilith and is illustrated in Figure 1. ES refers to the evaluation stack, which for StarLite is just the top of stack. F refers to the base of a module's code vector. Figure 2 lists the op code and function of each instruction.

As mentioned previously, StarLite is a pure stack architecture, while the Lilith is not. Thus, instructions were added to handle argument passing and procedure return. For example, a return-and-pop-arguments instruction was defined. For portability, we added instructions to load each constant type. As a result, differences in byte ordering or representation can be handled by the interpreter while leaving the compiler free to define a canonical object format.

<- 8 Bits ->




| MODE | EFFECTIVE OPERAND |
|---|---|
| Stack : | (ES++), operands are on the evaluation stack |
| Immediate: | A(-1..15), B(0..255), C(16 bits), D(32 bits) |
| Procedure Local : | ((L)+A) or ((L)+B) |
| Module Global : | ((G)+A) or ((G)+B) |
| External Module : | ((ModuleFrameTable[M])+N) |
| Indirect : | ((ES++)) |
| Indexed with NIL check : | ((ES++)+A) or ((ES++)+B) |
| Aligned Index with check : | ((ES++)*ElementSize + (ES++)) |
| String constant : | (((G)+2)+B) |
| PC Relative : | (F)+(PC)+B or (F)+(PC)+C |

Figure 1. The StarLite Op Code Format and Addressing Modes

| | 0 | 40 | 100 | 140 | 200 | 240 | 300 | 340 |
|---|---|---|---|---|---|---|---|---|
| 0 | LI0 | LLW | LGW | LSW0 | LSW | REST | FOR1 | MOV |
| 1 | LI1 | LLD | LGD | LSW1 | LSD | ILL | FOR2 | CMP |
| 2 | LI2 | LEW | LGW2 | LSW2 | LSD0 | DNEG | ENTC | ILL |
| 3 | LI3 | LED | LGW3 | LSW3 | ILL | DMOD | EXC | ILL |
| 4 | LI4 | LLW0 | LGW4 | LSW4 | LSTA | DABS | TRAP | ILL |
| 5 | LI5 | LLW1 | LGW5 | LSW5 | LXB | UCHK | CHK | ILL |
| 6 | LI6 | LLW2 | LGW6 | LSW6 | LXW | ROT | CHKZ | ILL |
| 7 | LI7 | LLW3 | LGW7 | LSW7 | LXD | SYS | CHKS | ILL |
| 10 | LI8 | LLW4 | LGW8 | LSW8 | DADD | ILL | EQL | GB |
| 11 | LI9 | LLW5 | LGW9 | LSW9 | DSUB | ILL | NEQ | GB1 |
| 12 | LI10 | LLW6 | LGW10 | LSW10 | DMUL | ULSS | LSS | ALOC |
| 13 | LI11 | LLW7 | LGW11 | LSW11 | DDIV | ULEQ | LEQ | ENTR |
| 14 | LI12 | LLW8 | LGW12 | LSW12 | CDBL | UGTR | GTR | RTN |
| 15 | LI13 | LLW9 | LGW13 | LSW13 | DCMP | UGEQ | GEQ | CX |
| 16 | LI14 | LLW10 | LGW14 | LSW14 | ILL | ILL | ABS | CI |
| 17 | LI15 | LLW11 | LGW15 | LSW15 | ILL | ILL | NEG | CF |
| 20 | LIB | SLW | SGW | SSW0 | SSW | SPW | OR | CL |
| 21 | LIR | SLD | SGD | SSW1 | SSD | SPD | XOR | CL1 |
| 22 | LIW | SEW | SGW2 | SSW2 | SSD0 | RTNS | AND | CL2 |
| 23 | LID | SED | SGW3 | SSW3 | ILL | LSSA | COM | CL3 |
| 24 | LLA | SLW0 | SGW4 | SSW4 | TS | LSSAC | IN | CL4 |
| 25 | LGA | SLW1 | SGW5 | SSW5 | SXB | COPD | LIN | CL5 |
| 26 | LSA | SLW2 | SGW6 | SSW6 | SXW | DECS | ILL | CL6 |
| 27 | LEA | SLW3 | SGW7 | SSW7 | SXD | PCOP | NOT | CL7 |
| 30 | JPC | SLW4 | SGW8 | SSW8 | FADD | UADD | ADD | CL8 |
| 31 | JP | SLW5 | SGW9 | SSW9 | FSUB | USUB | SUB | CL9 |
| 32 | JPFC | SLW6 | SGW10 | SSW10 | FMUL | UMUL | MUL | CL10 |
| 33 | ILL | SLW7 | SGW11 | SSW11 | FDIV | UDIV | DIV | CL11 |
| 34 | JPBC | SLW8 | SGW12 | SSW12 | FCMP | UMOD | ILL | CL12 |
| 35 | JPB | SLW9 | SGW13 | SSW13 | FABS | ILL | BIT | CL13 |
| 36 | ORJP | SLW10 | SGW14 | SSW14 | FNEG | SHL | NOP | CL14 |
| 37 | ANDJP | SLW11 | SGW15 | SSW15 | FFCT | SHR | ILL | CL15 |

Figure 2a. Op Code Table

Figure 2b. Op Code Definitions

| Op | Format | Traps | Operation |
|---|---|---|---|
| ADD SUB MUL DIV NEG ABS | | | INTEGER |
| | | stku, iovf, idivz | > (pop16), >, < (push16), + - * DIV u- ABS |
| DADD DSUB DMUL DDIV DNEG DMOD DABS | | | LONGINT |
| | | stku, dovf, ddivz | >>, >>, << (push32), + - * DIV u- MOD ABS |
| DCMP | | stku | if >> r >> then <<, = 0 0, lss 0 1, gtr 1 0 |
| EQL NEQ LSS LEQ GTR GEQ | | | INTEGER |
| | | stku | if > r > then < 1 else < 0 |
| FADD FSUB FMUL FDIV FNEG FABS | | | REAL |
| | | stku, fovf, funf, fdivz | >>, >>, <<, + - * / u- ABS |
| FCMP | | stku | if >> r >> then <<, = 0 0, lss 0 1, gtr 1 0 |
| FFCT | byte | stku, stko, fsig | b=0 FLOAT(LONGINT) b=1 TRUNC(LONGINT) b=2 TRUNC(REAL) b=3 LONGINT(CARDINAL) b=4 FLOAT(CARDINAL) |
| OR XOR AND COM | | stku | BITSET, >, >, ! xor & ~ |
| UADD USUB UMUL UDIV UMOD | | | CARDINAL |
| | | stku, covf, cdivz | >, >, <, + - * DIV MOD |
| ULSS ULEQ UGTR UGEQ | | | CARDINAL |
| | | stku | if > r > then < 1 else < 0 |
| ALOC | card,byte | stko | inc stack by c words; (L+wrdOffst b+frameSz):=adr |
| ANDJP | byte | stku | if > # 0 then PC += 1 else < 0; PC += b |
| BIT | | stku, covf | >i, < 2**i |
| CDBL | | stku, stko | convert top of stack from INTEGER to LONGINT |
| CF | byte | stko, missm, ilcall | like CX; the 16-bit mn,pn is b+1 words below S |
| CHK | | stku, subrange | >b, >a, >i, if (i lss a) or (i gtr b) then err else <i |
| CHKS | | stku, sign | >i, if i less 0 then err else <i |
| CHKZ | | stku, subrange | >b, >i, if i > b then err else <i |
| CI | byte-pn | stko, ilcall | call local-proc b; like CL but uses >> as static link |
| CL | byte-pn | stko, ilcall | call local-proc b; L,S=new frame; handler=NIL; staticL=oldL; L=oldL; PC=return offset in Code |
| CLi | | stko, ilcall | call local-proc i; L,S=new frame; handler=NIL; staticL=oldL; L=oldL; PC=return offset in Code |
| CMP | | stku, iladr | >i, >>s, >>d, cmp i words of (d) and (s) = <1 # <0 |
| COPD | | stku, stko | >>i, << i, <<i |
| CX | mn(8),pn(8) | stko, missm, ilcall | call ext m, proc p; saves G for caller; checks xCode > |
| DECS | | stku | |
| ENTC | card | stku, case | >cs, PC += c, next 2 wrds are lo,hi limit on all cases if (cs ge lo)&(cs le hi) PC += then (cs-lo+3)*2 else 4 PC += INTEGER(next 2 words) |
| ENTR | byte | stko | inc stack by b words for local variables |
| EXC | | stku, iladr | >byte offset, PC = F + b |
| FOR1 | byte,card | stku, iladr | >limit, >initial value, >>variableAdr if (b=0) and (init leq lim) or (b=1) and (init geq lim) (vadr):=init; <<vadr; <lim else PC += c |
| FOR2 | step-sb,int | stku, iladr, iovf | >limit, >>variableAdr if (sb geq 0) and (*vadr+sb leq lim) or (sb lss 0) and (*vadr+sb geq lim) then *vadr += sb; PC += int; <<vadr; <lim else fall through the loop |
| GB | byte | stko, iladr | << the static link b levels back, L if b=0 |

| Op | Format | Traps | Operation |
|---|---|---|---|
| GB1 | | stko, iladr | << L^.staticLink |
| ILL | | ilinst | illegal op code |
| IN | | stku, covf | >bs, >in, if bs & > 2**in then < 1 else < 0 |
| JP | integer | | PC += i |
| JPB | byte | | PC -= b |
| JPBC | byte | stku | if > = 0 then PC -= b else PC += 1 |
| JP(F)C | byte/card | stku | if > = 0 then PC += b/c else PC += 1/2 |
| LEA | byte-m,byte | stko, missm, iladr | << dft[m] + 8-bit word offset |
| L(S)ED | byte-m,byte | stku, stko, missm,iladr | << / >> (dft[m] + 8-bit word offset) |
| L(S)EW | byte-m,byte | stku, stko, missm,iladr | < / > (dft[m] + 8-bit word offset) |
| LIB | byte | stko | < zero-extended byte |
| LID | longint | stko | << LONGINT |
| LIi | | stko | < zero-extended i |
| LIN | | stko | < -1 |
| LIR | real | stko | << REAL |
| LIW | 16bits | stko | < 16 bits |
| LGA | byte | stko | << G + 8-bit word offset |
| L(S)GD | byte | stko, stku | << / >> (G + 8-bit word offset) |
| L(S)GW | byte | stko, stku | < / > (G + 8-bit word offset) |
| L(S)GWi | | stko, stku | < / > (G + word offset i) |
| LLA | byte | stko | << L + 8-bit word offset + frame size |
| L(S)LD | byte | stko, stku | << / >> (L + 8-bit word offset + frame size) |
| LSA | byte | stku, stko, iladr | >>adr, < adr + 8-bit word offset |
| L(S)SD | byte | stku, stko, iladr | >>adr, << / >> (adr + 8-bit word offset) |
| L(S)SD0 | | stku, stko, iladr | >>adr, << / >> (adr) |
| LSSA | | stku, iladr | >word offset, >> adr, << adr + w |
| LSSAC | byte | stku, iladr | >word offset j; (S - b-2 words) += j |
| LSTA | byte | stko, iladr | << G^.stringPointer + 8-bit word offset |
| L(S)LW | byte | stko, stku | < / > (L + 8-bit word offset + frame size) |
| L(S)LWi | | stko, stku | < / > (L + word offset i + frame size) |
| L(S)SWi | | stku, iladr | >>adr, < / > (adr + word offset i) |
| L(S)W | byte | stku, iladr | >>adr, < / > (adr + 8-bit word offset) |
| L(S)XB | | stku, iladr | >i, >>adr, < (adr + byte offset i) |
| L(S)XD | | stku, iladr | >i, >>adr, < (adr + doubleword offset i) |
| L(S)XW | | stku, iladr | >i, >>adr, < (adr + word offset i) |
| MOV | | stku, iladr | >i, >>s, >>d, move i words of (s) to (d) |
| NOP | | | no operation |
| NOT | | stku | if > = 0 then < 1 else < 0 |
| ORJP | byte | stku | if < = 0 then PC += 1 else > 1; PC += b |
| REST | | stku, iladr | >> frame adr; restore control to its event handler |
| ROT | byte | stku | b=1, >b >a <b; b=2, >c >b >a <b <c |
| RTN | | coend, iladr | proc return; restore the caller's frame |
| RTNS | byte | iladr | >argLstSize; b=wordsInResult; pop args; copy result |
| SHL | | stku, covf | >i, >c; < c * 2**i |
| SHR | | stku, covf | >i, >c; < c div 2**i |
| SPD | byte,byte | | 32(L + wrd off b1 + frame size) := (L - wrd offst b2) |
| SPW | byte,byte | | 16(L + wrd off b1 + frame size) := (L - wrd offst b2) |
| SYS | | | reserved for system functions |
| TRAP | | stku, i | >i; generate trap i |
| TS | | stku, iladr | >>ic; << (ic); (ic)=1 |
| UCHK | | stku, subrange | >hi, >lo, >a, if (a<lo) or (a>hi) then err else <a |

Other extensions involved changing from 16-bit to 32-bit addressing, adding an exception-handling instruction, creating additional arithmetic operators for the LONGINT and REAL types, and redefining offset-based addressing.

In the Lilith, there are special instructions to manipulate local variables. For instance, LLW6 loads the word at an offset of six from the L register. The problem with this definition is that the frame size may vary from one machine to another, or it may vary because of requirements dictated by a prototype operating system. In any case, any variation requires changing the compiler, which results in a loss of portability. The StarLite solution was to define all offsets as being relative to the first variable location that could be referenced. Thus, the interpreter adds the size of the control information to the supplied offset. This change affected the instructions that manipulated local, global, and external variables.

The deletions from the instruction set included instructions that dealt with the coroutine structure, interrupts, priority, or the DFT. In essence, the instruction set is independent of its execution environment since it is defined only in terms of its logical relationship to the components of a Modula-2 program. The underlying architecture is manipulated by using procedures that are exported from an extracode module. The opcode space that is made available by these changes is used to encode extensions, such as the LONGREAL type.

## 5. Summary

The StarLite system has been operational for a year. It is being used to develop operating systems, distributed database systems, and new network protocols. The architecture has been the "glue" that has enabled the other pieces of the environment to be put together in a way that maximizes a researcher's productivity.

While the initial version of the environment executes as a single UNIX process, future versions could take excellent advantage of both load balancing to distribute a running prototype across a number of machines and of multiprocessor support, such as is found in Mach or Taos.

We gratefully acknowledge the inspiration provided by the Lilith, which may have a longer life as a virtual machine than it ever had as a physical one.

## References

(1) Wirth, N., The Personal Computer Lilith, ETH Zurich, Institut fur Informatik Technical Report 40, (April 1981).

(2) Sweet, R. and J. Sandman, Static Analysis of the Mesa Instruction Set, *1st Symp. on Architectural Support for Programming Languages and Operating Systems*, (March 1982), 158-166.

(3) Canon, M.D. et al, A Virtual Machine Emulator for Performance Evaluation, *Communications of the ACM 23*, 2(Feb. 1980), 71-80.

(4) Rovner, P., Extending Modula-2 To Build Large, Integrated Systems, *IEEE Software 3*, 6(Nov. 1986) 46-57.

(5) Wirth, N., Microprocessor Architectures: A Comparison Based on Code Generation by Compiler, *Communications of the ACM 29*, 10(Oct. 1986) 978-994.

```
01: MODULE main;
02:  IMPORT x, InOut;


03:  PROCEDURE a();
04:    VAR r:INTEGER;
05:  BEGIN
06:    InOut.WriteString("in a");
07:    InOut.WriteLn();
08:    r= x.p(3);
09:  END a;

10: BEGIN
11:  InOut.WriteString("in main");
12:  InOut.WriteLn();
13:  a();
14: END main.
```

```
MODULE main          key DEA8 7 15
  data size 1A bytes  code size 34 bytes
IMPORT x             key D6D8 2 15
IMPORT InOut         key 9344 3AC AE73
GLOBAL DATA
        C:    i n \b a \0 \0
        12:   i n \b m a i n \0 \0
CODE
00 :    4     offset to module initialization
02 :    10    offset to procedure 'a'
04 :    LGA initializedFlag; TS; JPFC A; RTN
0A :    LGA StringArea; JP 21; NOP

10:     ENTR 1          (* procedure 'a' *)
12:     LSTA 0; LI3; CX 2 C; CX 2 B
1B:     LI3; CX 1 1; SLW0; RTN

21:     SGD stringPointer
23:     CX 1 0          (* init mod 1 (x); call proc 0 *)
26:     CX 2 0          (* init module 2 (InOut) *)
29:     LSTA 3; LI6; CX 2 C; CX 2 B
32:     CL1; RTN
FIXUP offsets 16 19 1D 24 27 2D 30
```

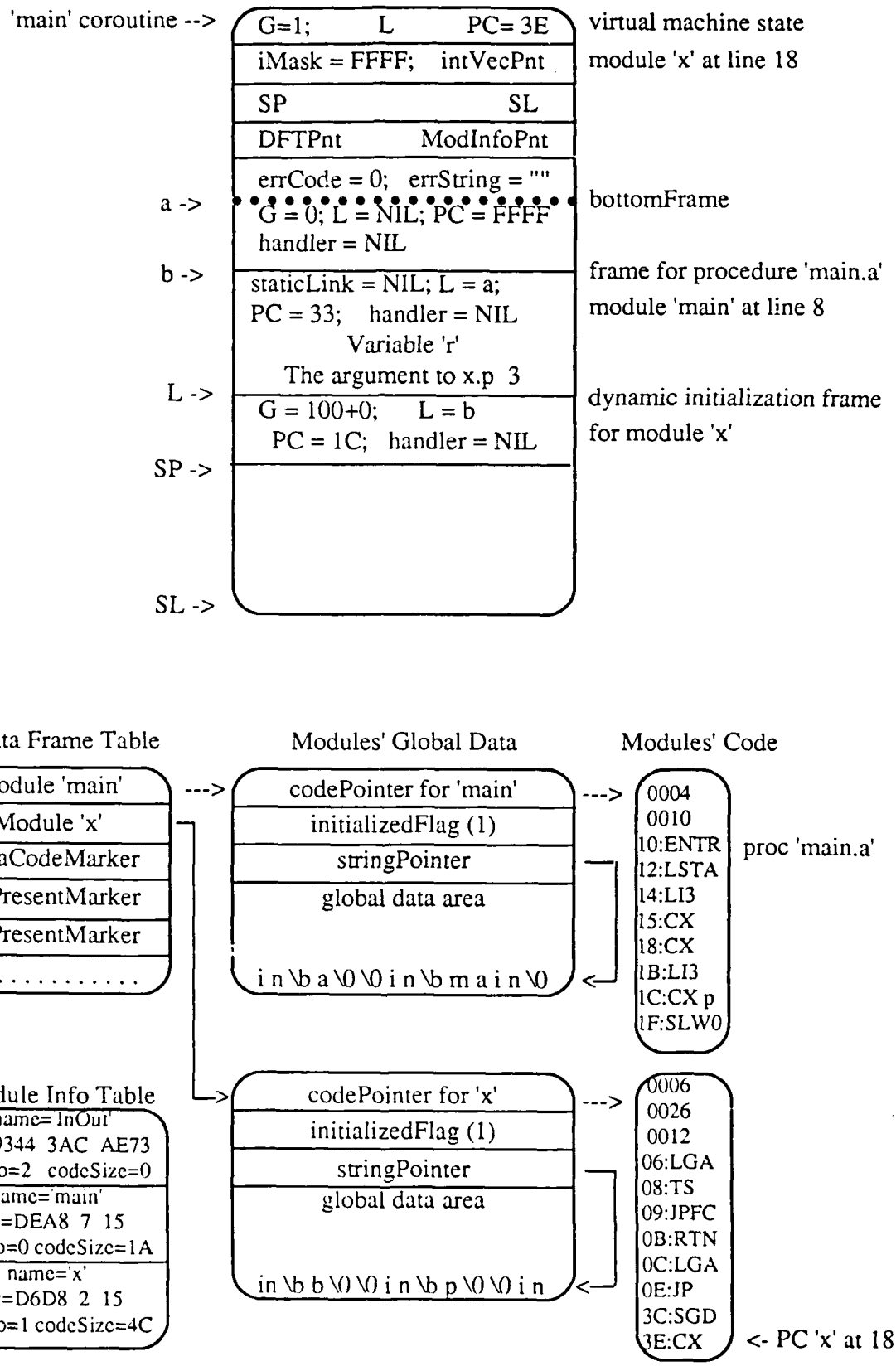
```
01: MODULE x;
02:  IMPORT InOut;


03:  PROCEDURE p(i:INTEGER):INTEGER;
04:    VAR t:INTEGER;
05:    PROCEDURE b():INTEGER;
06:      VAR s:INTEGER;
07:    BEGIN
08:      InOut.WriteString("in b");
09:      InOut.WriteLn();
10:      HALT;
11:      RETURN 3;
12:    END b;
13:  BEGIN
14:    InOut.WriteString("in p");
15:    InOut.WriteLn();
16:    RETURN b()*i;
17:  END p;

18: BEGIN
19:  InOut.WriteString("in x");
20:  InOut.WriteLn();
21: END x.
```

```
MODULE x                    key D6D8 2 15
  data size 1E bytes        code size 4C bytes
IMPORT InOut                key 9344 3AC AE73
GLOBAL DATA
        C:    i n \b b \0 \0
        12:   i n \b p \0 \0
        18:   i n \b x \0 \0
CODE
00 :    6     offset to module initialization
02 :    26    offset to procedure 'p'
04 :    12    offset to procedure 'b'
06 :    LGA initializedFlag; TS; JPFC C; RTN
0C :    LGA StringArea; JP 3C; NOP

12:     ENTR 1          (* procedure 'b' *)
14:     LSTA 0; LI3; CX 1 C; CX 1 B
1D:     LI1; TRAP; LI3; LI0; RTNS 1; LI5; TRAP; NOP
26:     ENTR 2; SPW 6 1       (* procedure 'p' *)
2B:     LSTA 3; LI3; CX 1 C; CX 1 B
34:     CL2; LLW6; MUL; LI1; RTNS 1; LI5; TRAP

3C:     SGD stringPointer
3E:     CX 1 0          (* init module 1 (InOut) *)
41:     LSTA 6; LI3; CX 1 C; CX 1 B
4A:     RTN
FIXUP offsets 18 1B 2F 32 3F 45 48
```

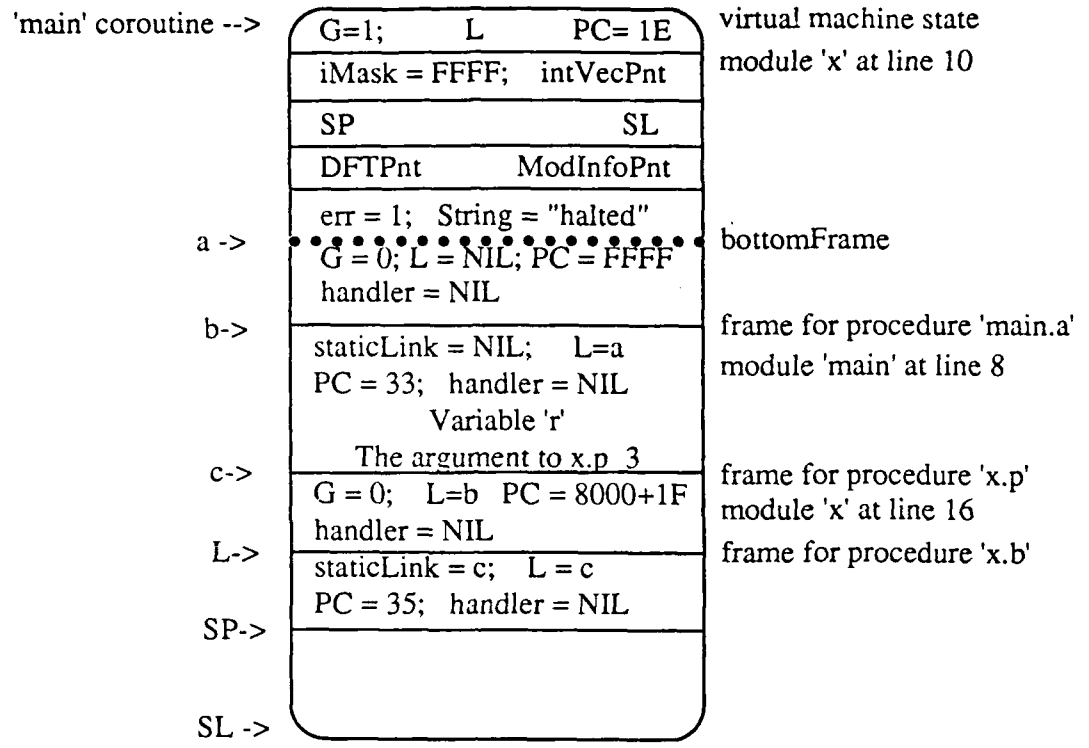
Appendix A. An Example Program and Its Object Modules

| | |
|---|---|
| CLi | 0<i<16, Call Local procedure number i |
| CX M N | 0<=M,N<=255, Call eXternal module M, procedure number N |
| ENTR N | 0<=N<=255, ENTeR procedure, reserves N words for local variables |
| JP N | -32768<=N<=32767, JumP to PC+N bytes |
| JPFC N | 0<=N<=255, Jump Forward Conditional N bytes only if the top of stack is zero |
| LIi | 0<=i<=15, Load Immediate i |
| LGA N | 0<=N<=255, Load the Global Address at location G+N words |
| LSTA N | 0<=N<=255, Load STring Address at stringPointer+N words |
| NOP | Null Operation |
| RTN | ReTurN from a subroutine |
| RTNS N | 0<=N<=255, ReTurN and adjust stack, N=size of return value, top-of-stack=size of argument list |
| SLWi | 0<=i<=11, Store top of stack in Local Word i |
| SGD N | 0<=N<=255, Store in Global word N the Doubleword at the top of stack |
| TRAP | generate the exception identified by the top of stack value |
| TS | Test and Set on the address contained in the top of stack |

## Appendix A1. An Explanation of the Example's Op Codes




Appendix A2. The Coroutine State During Dynamic Initialization


'main' coroutine -->
G=1; L PC= 1E
virtual machine state
iMask = FFFF; intVecPnt
module 'x' at line 10
SP SL
DFTPnt ModInfoPnt
err = 1; String = "halted"
a ->
G = 0; L = NIL; PC = FFFF
bottomFrame
handler = NIL
b->
staticLink = NIL; L=a
frame for procedure 'main.a'
PC = 33; handler = NIL
module 'main' at line 8
Variable 'r'
The argument to x.p 3
c->
G = 0; L=b PC = 8000+1F
frame for procedure 'x.p'
module 'x' at line 16
handler = NIL
L->
staticLink = c; L = c
frame for procedure 'x.b'
PC = 35; handler = NIL
SP->
SL ->
Data Frame Table
Modules' Global Data
Modules' Code
Module 'main'
codePointer for 'main'
0004
0010
10:ENTR
12:LSTA
14:LI3
15:CX
18:CX
1B:LI3
1C:CX p
1F:SLW0
proc 'main.a'
Module 'x'
initializedFlag (1)
extraCodeMarker
stringPointer
notPresentMarker
global data area
notPresentMarker
i n \b a \0 \0 i n \b m a i n \0
Module Info Table
name='InOut'
key= 9344 3AC AE73
modNo=2 codeSize=0
name='main'
key=DEA8 7 15
modNo=0 codeSize=1A
name='x'
key=D6D8 2 15
modNo=1 codeSize=4C
codePointer for 'x'
0006
0026
0012
12:ENTR
14:LSTA
16:LI3
17:CX
1A:CX
1D:LI1
1E:TRAP
1F:LI3
proc 'x.b'
<-PC (HALT)
initializedFlag (1)
stringPointer
global data area
in \b b \0 \0 i n \b p \0 \0 i n

DISTRIBUTION LIST

| Copy No. | |
|---|---|
| 1 - 6 | Director<br>Naval Research Laboratory<br>Washington, D.C. 20375<br>Attention: Code 2627 |
| 7 | Dr. James G. Smith, Code 1211<br>Applied Math and Computer Science Division<br>800 N. Quincy Street<br>Arlington, VA 22217-5000 |
| 8 - 19 | Defense Technical Information Center, S47031<br>Bldg. 5, Cameron Station<br>Alexandria, VA 22314 |
| 20 - 21 | Dr. R. P. Cook, CS |
| 22 - 23 | Dr. S. H. Son, CS |
| 24 | Dr. A. K. Jones, CS |
| 25 - 26 | Ms. E. H. Pancake, Clark Hall |
| 27 | SEAS Publications Files |
| * | Office of Naval Research Resident Representative<br>818 Connecticut Ave., N.W.<br>Eighth Floor<br>Washington, D.C. 20006<br>Attention: Mr. Michael McCracken<br>Administrative Contracting Officer |

* Send Copy of Cover Letter Only

1991:ald:DW212R

# An Adaptive Checkpointing Scheme for Distributed Databases with Mixed Types of Transactions

Sang Hyuk Son

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

## ABSTRACT

Recent study shows the possibility of having a checkpointing mechanism that does not interfere with the transaction processing, and yet achieves the global consistency of the checkpoints. Non-interfering checkpointing mechanisms, however, may suffer from the fact that the diverged computation needs to be maintained by the system until all of the transactions that are in progress when the checkpoint begins, come to completion. For database systems with many long-lived transactions that need long execution time, this requirement of maintaining diverged computation may make non-interfering checkpointing not practical. In this paper, we present a checkpointing algorithm that is non-interfering with transaction processing. It prevents the well-known "domino effect", and saves intermediate results of the transaction in an adaptive manner, managing effectively both short and long-lived transactions in the system.

## 1. Introduction

The need for having recovery mechanisms in database systems is well acknowledged. In spite of powerful database integrity checking mechanisms which detect errors and undesirable data, it is possible that some erroneous data may be included in the database. Furthermore, even with a perfect integrity checking mechanism, failures of hardware and/or software at the processing sites may destroy the consistency of the database. In order to cope with those errors and failures, database systems provide recovery mechanisms, and checkpointing is a technique frequently used in such recovery mechanisms.

The goal of checkpointing in database management systems is to save a consistent state of the database on a separate secure device. In case of a failure, the stored data can be used to restore the database to an earlier point in time. Checkpointing must be performed so as to minimize both the costs of performing checkpoints and the costs of recovering the database. If the checkpoint intervals are very small, too much time and resources are spent in checkpointing; if these intervals are large, too much time is spent in recovery. Since checkpointing is an effective method for maintaining consistency of database systems, it has been widely used and studied by many researchers[1, 2, 3, 5, 6, 8, 9, 11, 15, 16].

Since checkpointing is performed during normal operation of the system, the interference with transaction processing must be kept to a minimum. It is highly desirable that users are allowed to submit transactions while the checkpointing is in progress, and the transactions are executed in the system concurrently with the checkpointing process. A quick recovery from failures is also desirable to many applications of database systems. For achieving quick recovery, each checkpoint needs to be globally consistent so that a simple restoration of the latest checkpoint can bring the database to a consistent state. In distributed database systems these desirable properties of non-interference and global consistency make checkpointing complicated and increase the workload of the system.

Recently, the possibility of having a checkpointing mechanism that does not interfere with the transaction processing, and yet achieves the global consistency of the checkpoints, has been studied [2, 5, 19]. The motivation of non-interfering checkpointing is to improve the system availability, that is, the system must be able to execute user transactions concurrently with the checkpointing process. The basic principle behind non-interfering checkpointing mechanisms is to create a diverged computation of the system such that the checkpointing process can view a consistent state that could result by running to completion all of the transactions that are in progress when the checkpoint begins, instead of viewing a consistent state that actually occurs by suspending further transaction execution.

Non-interfering checkpointing mechanisms, however, may suffer from the fact that the diverged computation needs to be maintained by the system until all of the transactions that are in progress when the checkpoint begins, come to completion. This may not be a major concern for a database system in which all the transactions are relatively short, and hence can be executed in a short time period. However, for database systems with many long-lived transactions that need long execution time, a non-interfering checkpointing may not be practical because of the following reasons:

(1) It takes a long time to complete one non-interfering checkpoint, resulting in a high storage and processing overhead.

(2) If a crash occurs before reflecting the result of a long-lived transaction in the checkpoint, the system must re-execute the transaction from the beginning, wasting all the resources used for the initial execution of the transaction.

This work was partially supported by the Office of Naval Research under contract no. N00014-86-K-0245, and by the Jet Propulsion Laboratory under contract no. 957721 through Virginia Institute for Parallel Computation.



CH2550-2/88/0000/0528$01.00 © 1988 IEEE

In this paper, we present a checkpointing algorithm which manages effectively both short and long-lived transactions in the database system. Our checkpointing algorithm operates in two different modes: *global mode* and *local mode*. In the global mode of operation, the algorithm is non-interfering with transaction processing, and efficiently generates globally consistent checkpoints. In the local mode of operation, it prevents the well-known "domino effect", and saves intermediate results of the transaction. Furthermore, only a minimal number of processes are involved in the checkpointing. This paper is organized as follows. Section 2 introduces a model of computation used in this paper. Section 3 describes the algorithm for non-interfering checkpoint creation. Section 4 raises problems associated with non-interfering checkpoint creation, and presents an adaptive checkpointing algorithm as a possible solution. Section 5 presents an informal proof of the correctness of the algorithm. Section 6 discusses the practicality and the robustness of the algorithm, and describes the recovery methods associated with the algorithm. Section 7 concludes the paper.

## 2. Model of Computation

### 2.1. Data Objects and Transactions

A *database* consists of a set of data objects. Each data object has a *value* and represents the smallest unit of the database accessible to the user. All user requests for access to the database are handled by the *database system*. We consider a distributed database system implemented on a computing system where several autonomous computers (called *sites)* are connected via a communication network. The set of data objects in a distributed database system is partitioned among its sites. A database is said to be *consistent* if the values of data objects satisfy a set of assertions. The assertions that characterize the consistent states of the database are called the *consistency constraints* [4].

The basic units of user activity in database systems are *transactions*. Each transaction represents a complete and correct computation, i.e., if a transaction is executed alone on an initially consistent database, it would terminate in a finite time and produce correct results, leaving the database consistent. A transaction is the unit of consistency and hence, it must be *atomic*. By atomic, we mean that intermediate states of the database must not be visible outside the transaction, and all updates of a transaction must be executed in an all-or-nothing fashion. A transaction is said to be *committed* when it is executed to completion, and it is said to be *aborted* when it is not executed at all. When a transaction is committed, the output values are finalized and made available to all subsequent transactions. We assume that the database system runs a correct transaction control mechanism (e.g., atomic commit algorithm[17] and concurrency control algorithm[18]), and hence assures the atomicity and the serializability of transactions.

Each transaction has a time-stamp associated with it [10]. A time-stamp is a number that is assigned to a transaction when initiated and is kept by the transaction. Two important properties of time-stamps are (1) no two transactions have the same time-stamp, and (2) only a finite number of transactions can have a time-stamp less than that of a given transaction.

### 2.2. Failure Assumptions

A distributed database system can fail in many different ways, and it is almost impossible to make an algorithm which can tolerate all possible failures. In general, failures in distributed database systems can be classified as failures of *omission* or *commission* depending on whether some action required by the system specification was not taken or some action not specified was taken. The simplest failures of omission are *simple crashes* in which a site simply stops running when it fails. The hardest failures are *malicious runs* in which a site continues to run, but performs incorrect actions. Most real failures lie between these two extremes.

In this paper, we do not consider failures of commission such as the "malicious runs" type of failure. When a site fails, it simply stops running (fail-stop). When the failed site recovers, the fact that it has failed is recognized, and a recovery procedure is initiated. We assume that site failures are detectable by other sites. This can be achieved either by network protocols or by high-level time-out mechanisms in the application layer. We also assume that network partitioning never occurs. This assumption is reasonable for most local area networks and some long-haul networks.

## 3. Non-Interfering Checkpoint Creation

### 3.1. Motivation of Non-Interference

The motivation of having a checkpointing scheme which does not interfere with transaction processing is well explained in [2] by using the analogy of migrating birds and a group of photographers. Suppose a group of photographers observe a sky filled with migrating birds. Because the scene is so vast that it cannot be captured by a single photograph, the photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. Furthermore, it is desirable that the photographers do not disturb the process that is being photographed. The snapshots cannot all be taken at precisely the same instance because of synchronization problems, and yet they should generate a "meaningful" composite picture.

In a distributed database system, each site saves the state of the data objects stored at it to generate a local checkpoint. We cannot ensure that the local checkpoints are saved at the same instance, unless a global clock can be accessed by all the checkpointing processes. Moreover, we cannot guarantee that the global checkpoint, consisting of local checkpoints saved, is consistent. Non-interfering checkpointing algorithms are very useful for the situations in which a quick recovery as well as no blocking of transactions is desirable. Instead of waiting for a consistent state to occur, the non-interfering checkpointing approach constructs a state that would result by completing the transactions that are in progress when the global checkpoint begins.

In order to make each checkpoint globally consistent, updates of a transaction must be either included in the checkpoint completely or not at all. To achieve this, transactions are divided into two groups according to their relations to the current checkpoint: *after-checkpoint-transactions* (ACPT) and *before-checkpoint-transactions* (BCPT). Updates belonging to BCPT are included in the current checkpoint while those belonging to ACPT are not included. In a centralized

database system, it is an easy task to separate transactions for this purpose. However, it is not easy in a distributed environment. For the separation of transactions in a distributed environment, a special time-stamp which is globally agreed upon by the participating sites is used. This special time-stamp is called the *Global Checkpoint Number* (GCPN), and it is determined as the maximum of the *Local Checkpoint Numbers* (LCPN) through the coordination of all the participating sites.

An ACPT can be reclassified as a BCPT if it turns out that the transaction must be executed before the current checkpoint. This is called the *conversion* of transactions. The updates of a converted transaction are included in the current checkpoint.

### 3.2. The Algorithm

There are two types of processes involved in the execution of the algorithm: *checkpoint coordinator* (CC) and *checkpoint subordinate* (CS). The checkpoint coordinator starts and terminates the global checkpointing process. Once a checkpoint has started, the coordinator does not issue the next checkpoint request until the first one has terminated.

The variables used in the algorithm are as follows:

(1) *Local Clock* (LC): a clock maintained at each site which is manipulated by the clock rules of Lamport[10].

(2) *Local Checkpoint Number* (LCPN): a number determined locally for the current checkpoint.

(3) *Global Checkpoint Number* (GCPN): a globally unique number for the current checkpoint.

(4) CONVERT: a Boolean variable showing the completion of the conversion of all the eligible transactions at the site.

Our basic checkpointing algorithm, called CP1, works as follows:

(1) The checkpoint coordinator broadcasts a Checkpoint Request Message with a time-stamp $LC_{CC}$. The local checkpoint number of the coordinator is set to $LC_{CC}$, and the coordinator sets the Boolean variable CONVERT to false:

$$CONVERT_{CC} := \text{false}$$
$$LCPN_{CC} := LC_{CC}$$

All the transactions at the coordinator site with the time-stamps not greater than $LCPN_{CC}$ are marked as BCPT.

(2) On receiving a Checkpoint Request Message, the local clock of site m is updated and $LCPN_m$ is determined by the checkpoint subordinate as follows:

$$LC_m := \max(LC_{CC} + 1, LC_m)$$
$$LCPN_m := LC_m$$

The checkpoint subordinate of site m replies to the coordinator with $LCPN_m$, and sets the Boolean variable CONVERT to false:

$$CONVERT_m := \text{FALSE}$$

and marks all the transactions at the site m with the time-stamps not greater than $LCPN_m$ as BCPT.

(3) The coordinator broadcasts the GCPN which is decided by:

$$GCPN := \max(LCPN_n) \qquad n = 1,..,N$$

(4) For all sites, after LCPN is fixed, all the transactions with the time-stamps greater than LCPN are marked as temporary ACPT. If a temporary ACPT wants to update any data objects, those data objects are copied from the database to the buffer space of the transaction. When a temporary ACPT commits, updated data objects are not stored in the database as usual, but are maintained as *committed temporary versions* (CTV) of data objects. The data manager of each site maintains the permanent and temporary versions of data objects. When a read request is made for a data object which has committed temporary versions, the value of the latest committed temporary version is returned. When a write request is made for a data object which has committed temporary versions, another committed temporary version is created for it rather than overwriting the previous committed temporary version.

(5) When the GCPN is known, each checkpointing process compares the time-stamps of the temporary ACPT with the GCPN. Transactions that satisfy the following condition become BCPT; their updates are reflected into the database, and are included in the current checkpoint.

$$LCPN < \text{time-stamp}(T) \leq GCPN$$

The remaining temporary ACPT are treated as actual ACPT; their updates are not included in the current checkpoint. These updates are included in the database after the current checkpointing has been completed. After the conversion of all the eligible BCPT, the checkpointing process sets the Boolean variable CONVERT to true:

$$CONVERT := \text{true}$$

(6) Local checkpointing is executed by saving the state of data objects when there is no active BCPT and the variable CONVERT is true.

(7) After the execution of local checkpointing, the values of the latest committed temporary versions are used to replace the values of data objects in the actual database. Then, all committed temporary versions are deleted.

The above checkpointing algorithm essentially consists of two phases. The function of the first phase (steps 1 through 3) is the assignment of GCPN that is determined from the local clocks of the system. The second phase begins by fixing the LCPN at each site. This is necessary because each LCPN sent to the checkpoint coordinator is a candidate of the GCPN of the current checkpoint, and the committed temporary versions must be created for the data objects updated by ACPT. The notions of committed temporary

versions and conversion from ACPT to BCPT are introduced to assure that each checkpoint contains all the updates made by transactions with earlier time-stamps than the GCPN of the checkpoint.

When a participant receives a Transaction Initiation Message from the coordinator, it checks whether or not the transaction can be executed at this time. If the checkpointing process has already executed step 5 and time-stamp(T) ≤ GCPN, then a reject message is returned, and the transaction is aborted. Therefore in order to execute step 6, each checkpointing process only needs to check active BCPT at its own site, and yet the consistency of the checkpoint can be achieved.

### 3.3. Termination of the Algorithm

The algorithm described so far has no restriction on the method of arranging the execution order of transactions. With no restriction, however, it is possible that the algorithm may never terminate. In order to ensure that the algorithm terminates in a finite time, we must ensure that all BCPT terminate in a finite time, because local checkpointing in step 6 can occur only when there is no active BCPT at the site.

Termination of transactions in a finite time is ensured if the concurrency control mechanism gives priority to older transactions over younger transactions. With such a time-based priority, it is guaranteed that once a transaction $T_i$ is initiated, then $T_i$ is never blocked by subsequent transactions that are younger than $T_i$. The number of transactions that may block the execution of $T_i$ is finite because only a finite number of transactions can be older than $T_i$. Among older transactions which may block $T_i$, there must be the oldest transaction which will terminate in a finite time, since no other transaction can block it. When it terminates, the second oldest transaction can be executed, and then the third, and so on. Therefore, $T_i$ will be executed in a finite time. Since we have a finite number of BCPT when the checkpointing is initiated, all of them will terminate in a finite time, and hence the checkpointing itself will terminate in a finite time. Concurrency control mechanisms based on time-stamp ordering as in [18] can ensure the termination of transactions in a finite time.

## 4. Adaptive Checkpoint Creation

In the previous section, we have shown that the algorithm will terminate in a finite time by selecting appropriate concurrency control mechanisms. However, the amount of time necessary to complete one checkpoint cannot be bound in advance; it depends on the execution time of the longest transaction classified as a BCPT. It implies that the storage and processing cost of the checkpointing algorithm may become unacceptably high if a long-lived transaction is included in the set of BCPT. We discuss the practicality of the non-interfering checkpoints in Section 6. In addition to that, all the resources used for the execution of such a long-lived transaction would be wasted if the transaction must be re-executed from the beginning due to system failures.

In this section, we extend our checkpointing algorithm CP1 to solve these problems. We assume that each transaction must carry the mark with it, when initiated, which tells whether it is a normal transaction or a long-lived transaction. The threshold to separate two types of transactions is application-dependent. In general, transactions that need hours of execution can be considered as long-lived transactions.

The new checkpointing algorithm, called CP2, operates in two different modes: *global mode* and *local mode*. The global mode operation of CP2 is basically the same as CP1, and it will efficiently generate consistent checkpoints in a non-interfering manner. In the local mode of operation, CP2 provides a mechanism to save consistent states of a transaction so that the transaction can resume execution from its most recent checkpoint.

As in the algorithm CP1, the checkpoint coordinator begins the algorithm CP2 by sending out Checkpoint Request Messages. Upon receiving this request message, each site checks whether any long-lived transaction is being executed at the site. If yes, the site reports it to the coordinator, instead of sending LCPN. Otherwise (i.e., no long-lived transaction in the system), CP2 continues the same procedure as CP1. If any site reports the existence of long-lived transaction, the coordinator switches to the local mode of operation, and informs each site to operate in the local mode. The checkpoint coordinator sends Checkpoint Request Messages to each site at an appropriate time interval to initiate the next checkpoint in the global mode. This attempt will succeed if there is no active long-lived transactions in the system.

In the local mode of operation, each long-lived transaction is checkpointed separately from other long-lived transactions. The coordinator of the long-lived transaction initiates the checkpoint by sending Checkpoint Request Messages to its participants. A checkpoint at each site saves a local state of a long-lived transaction. For satisfying the correctness requirement, a set of checkpoints, one per each participating site of a global long-lived transaction, should reflect the consistent state of the transaction. Inconsistent set of checkpoints may result by non-synchronized execution of associated checkpoint. For example, consider a long-lived transaction T being executed at sites P and Q, and a checkpoint taken at site P at time X, and at site Q at time Y. If a message M is sent from P after X, and received at Q before Y, then the checkpoints would save the reception of M but not the sending of M, resulting in a checkpoint representing an inconsistent state of T.

We use message numbers for achieving consistency in a set of local checkpoints of a long-lived transaction. Messages that are exchanged by the participating transaction managers of a long-lived transaction contains a message number tag. Transaction managers of a long-lived transaction use monotonically increasing numbers in the tag of its outgoing messages, and each maintains the tag numbers of the last message it received from other participants. On receiving a checkpoint request, a participant compares the message number attached to the request message with the tag number it received last from the coordinator. The participant replies OK to the coordinator and executes local checkpointing only if the request tag number is not less than the number it has maintained. Otherwise, it reports to the coordinator that the checkpointing cannot be executed with that request message.

If all the replies from the participants arrive and are all OK, the coordinator decides to make all the local checkpoints permanent. Otherwise, the decision is to discard the current

checkpoint, and to initiate a new checkpoint. This decision is delivered to all participants. After a new permanent checkpoint is taken, any previous checkpoints will be discarded at each site.

## 5. Consistency of Global Checkpoints

In this section we give an informal proof of the correctness of the algorithm. We show that each mode of operation satisfies the requirement of correctness. Although the consistency is our correctness criteria for the checkpointing algorithm, the unit for consistency is different for different mode of operation; a transaction is the unit of consistency in the global mode, while an event of a transaction is the unit of consistency in the local mode. We first show the consistency in the global mode.

### 5.1. Consistency in Global Mode

In addition to proving the consistency of the checkpoints generated by the algorithm in the global mode, we show that the algorithm has another desirable property that each checkpoint contains all the updates of transactions with earlier time-stamps than its GCPN. This property reduces the work required in the actual recovery, which is discussed in Section 6. A longer and more thorough discussion on the correctness of the algorithm is given in [19].

The properties of the algorithm we want to show are

(1) a set of all local checkpoints with the same GCPN represents a consistent database state, and

(2) all the updates of the committed transactions with earlier time-stamps than the GCPN are reflected in the current checkpoint.

Note that only one checkpointing process can be active at a time because the checkpointing coordinator is not allowed to issue another checkpointing request before the termination of the previous one.

A database state is consistent if the set of data objects satisfies the consistency constraints[4]. Since a transaction is the unit of consistency, a database state S is consistent if the following holds:

(1) For each transaction T, S contains all subtransactions of T or it contains none of them.

(2) If T is contained in S, then each predecessor T' of T is also contained in S. (T' is a predecessor of T if it modified the data object which T accessed at some later point in time.)

For a set of local checkpoints to be globally consistent, all the local checkpoints with the same GCPN must be consistent with each other concerning the updates of transactions that are executed before and after the checkpoint. Therefore, to prove that the algorithm satisfies both properties, it is sufficient to show that the updates of a global transaction T are included in $CP_i$ at each participating site of T, if and only if time-stamp(T) $\leq$ GCPN($CP_i$). This is enforced by the mechanism to determine the value of the GCPN, and by the conversion of the temporary ACPT into BCPT.

A transaction is said to be *reflected* in data objects if the values of data objects represent the updates made by the transaction. We assume that the database system provides a reliable mechanism for writing into the secondary storage such that a writing operation of a transaction is atomic and always successful when the transaction commits. Because updates of a transaction are reflected in the database only after the transaction has been successfully executed and committed, partial results of transactions cannot be included in checkpoints.

The checkpointing algorithm assures that the sequence of actions are executed in some specific order. At each site, conversion of eligible transactions occurs after the GCPN is known, and local checkpointing cannot start before the Boolean variable CONVERT becomes true. CONVERT is set to false at each site after it determines the LCPN, and it becomes true only after the conversion of all the eligible transactions. Thus, it is not possible for a local checkpoint to save the state of the database in which some of the eligible transactions are not reflected because they remain unconverted.

We can show that a transaction becomes BCPT if and only if its time-stamp is not greater than the current GCPN. This implies that all the eligible BCPT will become BCPT before local checkpointing begins in step 6. Therefore, updates of all BCPT are reflected in the current checkpoint.

From the atomic property of transactions provided by the transaction control mechanism (e.g. commit protocol in [17]), it can be assured that if a transaction is committed at a participating site then it is committed at all other participating sites. Therefore if a transaction is committed at one site, and if it satisfies the time-stamp condition above, its updates are reflected in the database and also in the current checkpoint at all the participating sites.

### 5.2. Consistency in Local Mode

In order to prove that the algorithm CP2 is correct in the local mode of operation, we need to show that a set of local checkpoints always represents a consistent state of the transaction that is checkpointed. In other words, it is sufficient to show that if the set of local checkpoints is consistent before the execution of CP2, the set of checkpoints is also consistent after the completion of CP2.

Since the initiation point of a transaction is consistent, the system has at least one set of consistent checkpoints of a transaction (i.e., the initiation point). Therefore, if CP2 does not generate a new set of checkpoints upon its termination, the system has the previous checkpoint which is consistent.

Without loss of generality, assume a new set of checkpoints is taken by CP2. We show by contradiction that the set of checkpoints after the termination of CP2 is consistent. Suppose it is not consistent. Then there are two transaction managers P and Q such that P sent Q a message M after making its checkpoint, and Q received M before making its checkpoint. Consider the case that P is the coordinator. Upon receiving a request message from the coordinator, Q must have sent OK because Q could not have made its checkpoint permanent otherwise. It implies that the tag number of the request message is greater than those of messages Q has received, a contradiction. If Q is the coordinator, P cannot start local checkpointing before receiving a request message from Q. Since Q sent the request message after receiving M, P must have received it after it sent M, a contradiction.

## 6. Discussion

The desirable properties of non-interference and global consistency not only make the checkpointing more complicated in distributed database systems, but also increase the workload of the system. It may turn out that the overhead of the checkpointing mechanism is unacceptably high, in which case the mechanism should be abandoned in spite of its desirable properties. The practicality of non-interfering checkpointing, therefore, depends partially on the amount of extra workload incurred by the checkpointing mechanism. In this section we consider practicality of non-interfering checkpointing algorithms, and discuss the robustness and recovery methods associated with the algorithm CP2.

### 6.1. Practicality of Non-interfering Checkpoints

There are two performance measures that can be used in discussing the practicality of non-interfering checkpointing: extra storage and extra workload required. The extra storage requirement of the algorithm is simply the committed temporary version (CTV) file size, which is a function of the expected number of ACPT of the site, the number of data objects updated by a typical transaction, and the size of the basic unit of information:

$$\text{CTV file size} = N_A \times (\text{number of updates}) \times (\text{size of the data object})$$

where $N_A$ is the expected number of ACPT of the site.

The size of the CTV file may become unacceptably large if $N_A$ or the number of updates becomes very large. Unfortunately, they are determined dynamically from the characteristics of transactions submitted to the database system, and hence cannot be controlled. Since $N_A$ is proportional to the execution time of the longest BCPT at the site, it would become unacceptably large if a long-lived transaction is being executed when a checkpoint begins at the site. The only parameter we can change in order to reduce the CTV file size is the granularity of a data object. The size of the CTV file can be minimized if we minimize the size of the data object. By doing so, however, the overhead of normal transaction processing (e.g., locking and unlocking, deadlock detection, etc) will be increased. Also, there is a trade-off between the degree of concurrency and the lock granularity[14]. Therefore the granularity of a data object should be determined carefully by considering all such trade-offs, and we cannot minimize the size of the CTV file by simply minimizing the data object granularity.

There is no extra storage requirement in intrusive checkpointing mechanisms[1, 8, 15]. However this property is balanced by the cases in which the system must block ACPT or abort half-way done global transactions because of the checkpointing process.

The extra workload imposed by the algorithm mainly consists of the workload for (1) determining the GCPN, (2) committing ACPT (move data objects to the CTV file), (3) reflecting the CTV file (move committed temporary versions from the CTV file to the database), and (4) making the CTV file clear when the reflect operation is finished. Among these, workload for (2) and (3) dominates others. As in extra storage estimation, they are determined by the number of ACPT and the number of updates. Therefore, as far as the values of these variables can be maintained within a certain threshold level, non-interfering checkpointing would not severely degrade the performance of the system. A detailed discussion on the practicality of non-interfering checkpointing is given in [19].

### 6.2. Site Failures

So far, we assumed that no failure occurs during checkpointing. This assumption can be justified if the probability of failures during a single checkpoint is extremely small. However, it is not always the case, and we now consider the method to make the algorithm resilient to failures.

During the global mode of operation, the algorithm CP2 is insensitive to failures of subordinates. If a subordinate fails before the broadcast of a Checkpoint Request Message, it is excluded from the next checkpoint. If a subordinate does not send its LCPN to the coordinator, it is excluded from the current checkpoint. When the site recovers from the failure, the recovery manager of the site must find out the GCPN of the latest checkpoint. After receiving information of transactions which must be executed for recovery, the recovery manager brings the database up to date by executing all the transactions whose time-stamps are not greater than the latest GCPN. Other transactions are executed after the state of the data objects at the site is saved by the checkpointing process.

An atomic commit protocol guarantees that a transaction is aborted if any participant fails before it sends a Precommit message to the coordinator. Therefore, site failures during the execution of the algorithm cannot affect the consistency of checkpoints because each checkpoint reflects only the updates of committed BCPT.

In the local mode of operation, a failure of a participant prevents the coordinator from receiving OK from all the participants, or prevents the participants from receiving the decision message from the coordinator. However, because a transaction is aborted by an atomic commit protocol, it is not necessary to make checkpointing robust to failures of participants.

The algorithm is, however, sensitive to failures of the coordinator. In particular, if the coordinator crashes during the first phase of the global mode of operation (i.e., before the GCPN message is sent to subordinates), every transaction becomes ACPT, requiring too much storage for committed temporary versions.

One possible solution to this involves the use of a number of *backup* processes; these are processes that can assume responsibility for completing the coordinator's activity in the event of its failure. These backup processes are in fact checkpointing subordinates. If the coordinator fails before it broadcasts the GCPN message, one of the backups takes the control. A similar mechanism is used in SDD-1 [7] for reliable commitment of transactions. Proper coordination among the backup processes is crucial here. In the event of the failure of the coordinator, one, and only one backup process has to assume the control. The algorithm for accomplishing this assumes an ordering among the backup processes, designated in order as $p_1, p_2, \ldots, p_n$. Process $p_{k-1}$ is referred to as the *predecessor* of process $p_k$ (for $k > 0$), and the coordinator is taken as the predecessor of process $p_1$.

We assume that the network service enables processes to be informed when a given site achieves a specified status

(simply UP or DOWN in this case). Initially, each of the backup processes checks the failure of its predecessor. Then the following rules are used.

(1) If the predecessor is found to be down, then the process begins to check the predecessor of the failed process.

(2) If the coordinator is found to be down, the first backup process assumes the control of checkpointing.

(3) If a backup process recovers, it ceases to be a part of the current checkpointing.

(4) After each checkpoint, the list of backup processes is adjusted by including all the UP sites.

These rules guarantee that at most one process, either the coordinator or one of the backup processes, will be in control at any given time. Thus a checkpointing will terminate in a finite time once it begins.

### 6.3. Recovery

The recovery from site crashes is called the *site recovery*. The complexity of the site recovery varies in distributed database systems according to the failure situation[15]. If the crashed site has no replicated data objects and if all the recovery information is available at the crashed site, local recovery is enough. Global recovery is necessary because of failures which require the global database to be restored to some earlier consistent state. For instance, if the transaction log is partially destroyed at the crashed site, local recovery cannot be executed to completion.

When a global recovery is required, the database system has two alternatives: a *fast* recovery and a *complete* recovery. A fast recovery is a simple restoration of the latest global checkpoint. Since each checkpoint generated by the algorithm is globally consistent, the restored state of the database is assured to be consistent. However, all the transactions committed during the time interval from the latest checkpoint until the time of crash would be lost. A complete recovery is performed to restore as many transactions that can be redone as possible. The trade-offs between the two recovery methods are the recovery time and the number of transactions saved by the recovery.

Quick recovery from failures is critical for some applications of distributed database systems which require high availability (e.g., *ballistic missile defense or air traffic control*). For those applications, the fate of the mission, or even the lives of human beings, may depend on the correct values of the data and the accessibility to it. Availability of a consistent state is of primary concern for them, not the most up-to-date consistent state. If a simple restoration of the latest checkpoint could bring the database to a consistent state, it may not be worthwhile to spend time in recovery by executing a complete recovery to recover some of the transactions.

For the applications in which each committed transaction is so important that the most up-to-date consistent state of the database is highly desirable, or if the checkpoint intervals are large such that a lot of transactions may be lost by the fast recovery, a complete recovery is appropriate to use. The cost of a complete recovery is the increased recovery time which reduces the availability of the database. Searching through the transaction log is necessary for a complete recovery. The second property of the algorithm (i.e., each checkpoint reflects all the updates of transactions with earlier time-stamps than its GCPN) is useful in reducing the amount of searching because the set of transactions whose updates must be redone can be determined by the simple comparison of the time-stamps of transactions with the GCPN of the checkpoint. Complete recovery mechanisms based on the special time-stamp of checkpoints (e.g., GCPN) have been proposed in [9, 20].

After the site recovery is completed using either a fast recovery procedure or a complete recovery procedure, the recovering site checks whether it has completed a local mode checkpointing for any long-lived transactions. If any local mode checkpoints are found, those transactions can be restarted from the saved checkpoints. In this case, the coordinator of the transaction requests all the participants to restart from their checkpoints if and only if they all are able to restart from that checkpoint. The coordinator makes a decision whether to restart the transaction from the checkpoint or from the beginning based on the responses from the participants, and sends the decision message to all the participants. We provide such a two-phase recovery protocol in order to maintain the consistency of the database in case of damaged checkpoints at the failure site. A transaction will be restarted from the beginning if any participant is not able to restore the checkpointed state of the transaction for any reason.

## 7. Concluding Remarks

During normal operation of the database system, checkpointing is performed to save information necessary for recovery from a failure. For better recoverability and availability of distributed database systems, checkpointing must be able to generate a globally consistent database state, without interfering with transaction processing. Site autonomy in distributed database systems makes the checkpointing more complicated than in centralized database systems. Also, long-lived transactions may substantially increase the overhead associated with non-interfering checkpointing, and make it unacceptable in many applications of the distributed systems. In this paper, a new checkpointing algorithm for distributed database systems is presented and discussed. The correctness of the algorithm is shown, and the robustness of the algorithm and recovery procedures associated with it are discussed. For the applications in which the system must execute a mixture of short and long-lived transactions, and the ability of continuous processing of transactions is so critical that the blocking of transaction activity for checkpointing is not feasible, we believe that the algorithm presented in this paper provides a practical solution to the problem of checkpointing and recovery in distributed database systems.

## REFERENCES


[1] Attar, R., Bernstein, P. A. and Goodman, N., Site Initialization, Recovery, and Backup in a Distributed Database System, IEEE Trans. on Software Engineering, November 1984, pp 645-650.

[2] Chandy, K. M., Lamport, L., Distributed Snapshots: Determining Global States of Distributed Systems,

ACM Trans. on Computer Systems, February 1985, pp 63-75.

[3] Dadam, P. and Schlageter, G., Recovery in Distributed Databases Based on Non-synchronized Local Checkpoints, Information Processing 80, North-Holland Publishing Company, Amsterdam, 1980, pp 457-462.

[4] Eswaran, K. P. et al, The Notion of Consistency and Predicate Locks in a Database System, Commun. of ACM, Nov. 1976, pp 624-633.

[5] Fischer, M. J., Griffeth, N. D. and Lynch, N. A., Global States of a Distributed System, IEEE Trans. on Software Engineering, May 1982, pp 198-202.

[6] Gelenbe, E., On the Optimum Checkpoint Interval, JACM, April 1979, pp 259-270.

[7] Hammer, M. and Shipman, D., Reliability Mechanisms for SDD-1: A System for Distributed Databases, ACM Trans. on Database Systems, December 1980, pp 431-466.

[8] Jouve, M., Reliability Aspects in a Distributed Database Management System, Proc. of AICA, 1977, pp 199-209.

[9] Kuss, H., On Totally Ordering Checkpoints in Distributed Databases, Proc. ACM SIGMOD, 1982, pp 293-302.

[10] Lamport, L., Time, Clocks and Ordering of Events in Distributed Systems, Commun. ACM, July 1978, pp 558-565.

[11] McDermid, J., Checkpointing and Error Recovery in Distributed Systems, Proc. 2nd International Conference on Distributed Computing Systems, April 1981, pp 271-282.

[12] Mohan, C., Strong, R., and Finkelstein, S., Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement Within Clusters of Processors, Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing, August 1983.

[13] Ricart, G. and Agrawala, A. K., An Optimal Algorithm for Mutual Exclusion in Computer Networks, Commun. of ACM, Jan. 1981, pp 9-17.

[14] Ries, D., The Effect of Concurrency Control on The Performance of A Distributed Data Management System, 4th Berkeley Conference on Distributed Data Management and Computer Networks, Aug. 1979, pp 221-234.

[15] Schlageter, G. and Dadam, P., Reconstruction of Consistent Global States in Distributed Databases, International Symposium on Distributed Databases, North-Holland Publishing Company, INRIA, 1980, pp 191-200.

[16] Shin, K. G., Lin, T.-H., Lee, Y.-H., Optimal Checkpointing of Real-Time Tasks, 5th Symposium on Reliability in Distributed Software and Database Systems, January 1986, pp 151-158.

[17] Skeen, D., Nonblocking Commit Protocols, Proc. ACM SIGMOD International Conference on Management of Data, 1981, pp 133-142.

[18] Son, S. H., On Multiversion Replication Control in Distributed Systems, Computer Systems Science and Engineering, Vol. 2, No. 2, April 1987, pp 76-84.

[19] Son, S. H. and Agrawala, A., Practicality of Non-Interfering Checkpoints in Distributed Database Systems, Proceedings of IEEE Real-Time Systems Symposium, New Orleans, Louisiana, December 1986, pp 234-241.

[20] Son, S. H. and Agrawala, A., An Algorithm for Database Reconstruction in Distributed Environments, 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 1986, pp 532-539.

## UNIVERSITY OF VIRGINIA
### School of Engineering and Applied Science

The University of Virginia's School of Engineering and Applied Science has an undergraduate enrollment of approximately 1,500 students with a graduate enrollment of approximately 560. There are 150 faculty members, a majority of whom conduct research in addition to teaching.

Research is a vital part of the educational program and interests parallel academic specialties. These range from the classical engineering disciplines of Chemical, Civil, Electrical, and Mechanical and Aerospace to newer, more specialized fields of Biomedical Engineering, Systems Engineering, Materials Science, Nuclear Engineering and Engineering Physics, Applied Mathematics and Computer Science. Within these disciplines there are well equipped laboratories for conducting highly specialized research. All departments offer the doctorate; Biomedical and Materials Science grant only graduate degrees. In addition. courses in the humanities are offered within the School.

The University of Virginia (which includes approximately 2,000 faculty and a total of full-time student enrollment of about 16,400), also offers professional degrees under the schools of Architecture, Law, Medicine, Nursing, Commerce, Business Administration, and Education. In addition, the College of Arts and Sciences houses departments of Mathematics, Physics, Chemistry and others relevant to the engineering research program. The School of Engineering and Applied Science is an integral part of this University community which provides opportunities for interdisciplinary work in pursuit of the basic goals of education, research, and public service.